UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Critérios Restritos de Teste de Software: Uma Contribuição Para Gerar Dados de Teste Mais Eficazez

Autor: **Silvia Regina Vergilio**

Orientador: **Prof. Dr. José Carlos Maldonado**

Co-orientador: **Prof. Dr. Mario Jino**

Dissertação submetida à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como requisito parcial para obtenção do Título de Doutor em Engenharia Elétrica.

27 de Julho de 1997

Aos meus pais, Wilson e Maria Elvira,
com amor e admiração.

Agradecimentos

Gostaria de agradecer ao meu orientador Prof. Dr. José Carlos Maldonado pela orientação, sugestões e críticas dadas durante o desenvolvimento deste trabalho. Pelo otimismo e amizade.

Ao Prof. Dr. Mario Jino pelo apoio, pela orientação e oportunidades oferecidos.

A todo grupo de Testes da Unicamp pela presença e contribuições nos seminários, pelo companheirismo demonstrado, e pelos cafezinhos. A Adalberto N. Crespo, Inês A.G. Boaventura, Cristina L.F. Moreira, Ivan Granja e Nelson C. Mendes. Ao Edmundo S. Spoto pela alegria, ao Paulo M.S. Bueno pela tranquilidade, a Plinio R.S. Vilela pela companhia em Purdue. À Leticia Mara Peres pelo entusiasmo.

A todos os amigos e colegas da Faculdade de Engenharia Elétrica da Unicamp e do Instituto de Ciências Matemáticas de São Carlos, USP, pelo excelente ambiente de trabalho. À Lucimara Desiderá, Claudia A. Tambascia e Raquel Schulze, pela amizade. Ao Bruno Schulze pelas sugestões dadas ao trabalho.

Aos professores da Universidade Federal do Paraná, especialmente à Cristina D. Murta pelo incentivo, Tânia M. Preto pelo bom humor, e Olga R. Belon pela força.

Às famílias "Delamaro" e "Maldonado" pelo apoio dado em Purdue.

Aos amigos José A. Carrilho e Raimundo Claudio pelas alegrias do dia a dia. À Maria Inês Vale pelo divertimento. Ao Claudio Barberato pelo carinho. Ao Renato Tutumi e em especial à Maria Adela R. Alvarez pela amizade e compreensão.

À minha família que sempre deu a maior força.

À CAPES e CNPq pelo apoio financeiro.

Resumo

Critérios de teste estrutural dividem o domínio de entrada de um programa em teste, em sub-domínios e requerem que pelo menos um ponto de cada sub-domínio seja executado, auxiliando na geração de dados de teste; permitem ainda, a avaliação da adequação de um dado conjunto de dados (casos) de teste. Uma vez particionado o domínio, é necessário responder à seguinte questão: "Que pontos de cada sub-domínio devem ser selecionados?". Isso diz respeito à tarefa de geração de dados de teste para satisfazer um critério. Essa é uma atividade bastante complexa de ser automatizada pois não existe um algoritmo de propósito geral para determinar um conjunto de casos de teste que satisfaça um dado critério para um particular programa. Não é possível nem mesmo determinar se esse conjunto existe. Na literatura são encontradas diferentes técnicas de geração de dados de teste que utilizam diferentes fundamentos para selecionar pontos do domínio que descrevem certos tipos de erros e, por isso, com alta probabilidade de revelar esses erros. No entanto, essas técnicas são apresentadas de forma dissociada dos critérios estruturais.

Este trabalho introduz uma família de Critérios Baseados em Restrições, denominados Critérios Restritos, que têm o objetivo de aumentar a eficácia das atividades de teste e de oferecer medidas de cobertura. Os Critérios Restritos permitem a utilização de critérios estruturais juntamente com os princípios de técnicas de geração de dados de teste sensíveis a erros e foram motivados por resultados de estudos teóricos e empíricos conduzidos com essas técnicas. Nesse trabalho, esses resultados, que serviram como motivação para a introdução dos Critérios Restritos, são apresentados. São discutidos aspectos de complexidade e de relação de inclusão entre os Critérios Restritos e os demais critérios. Também é proposta uma extensão da ferramenta de testes POKE-TOOL para apoiar a utilização desses critérios e para facilitar a etapa de geração de dados de teste.

Um experimento de avaliação dos Critérios Restritos é descrito. Os resultados desse experimento comprovam a aplicabilidade desses critérios e indicam um aumento no número de erros revelados. Ao final, são propostas duas estratégias de geração de dados de teste para satisfazer critérios de teste estrutural. Elas têm como objetivo reduzir os efeitos causados por caminhos não executáveis na atividade de teste e gerar dados com alta probabilidade de revelar erros. Entre essas estratégias propõe-se uma estratégia incremental, baseada na hierarquia entre os critérios, e que garante a preservação da relação de inclusão mesmo quando o fator eficácia é considerado.

Abstract

Structural testing criteria divide the program input domain to sub-domains and require the execution of at least one point from each sub-domain. They support the test data generation phase and the adequacy analysis of a test set.

Once the domain is divided, the question is posed: "What points in each sub-domain should be selected?" This question concerns to the task of generating test data to satisfy a criterion, which is very complex to be automated since there is no general algorithm to determine a set of test cases that satisfy a given criterion; it is not possible to determine even that such set exists. In the literature there are different test data generation techniques with different principles for choosing points from the program domain associated to certain errors, with a high probability of revealing them. However these techniques are presented not associated to structural criteria.

This work introduces a family of criteria, named Constraint Based Criteria. They have the goal of increasing the testing activity efficacy and make possible to obtain coverage measures. They permit the use of error-sensitive data generation techniques with structural criteria. The Constraint Based Criteria proposal was motivated by results obtained from empirical and theoretical studies with these techniques. These results are presented. Aspects of complexity and the inclusion relation among Constraint Based Criteria and other criteria are discussed. An extension to POKE-TOOL is proposed with the goal of supporting Constraint Based Criteria and easing the data test generation phase.

An experiment to evaluate the Constraint Based Criteria is described. The results from this experiment show their applicability and an increase in the number of revealed errors. And finally, two test data generation strategies to satisfy structural criteria are proposed. They have the goal of reducing the effects of infeasible paths in the testing activity and of generating test data with high probability of revealing errors. One of these strategies is incremental, based on the hierarchy of criteria and always preserve the inclusion relation among criteria even when the factor efficacy is considered.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contexto

A importância da disciplina de Engenharia de Software tem se tornado cada vez maior com o aumento da utilização dos produtos de software em tarefas essenciais às atividades humanas; através da proposição de métodos, ferramentas e procedimentos a engenharia de software tem contribuído para aumentar a produção de software e, sobretudo, a sua qualidade.

Dentre as atividades de garantia e avaliação da qualidade de software, a atividade de teste é fundamental [Pre92]. Isso porque o processo de desenvolvimento de software está sujeito a diversos tipos de erros e não se pode garantir que as demais fases do projeto foram realizadas adequadamente. No entanto, teste é uma atividade cara, que requer esforço e tempo [Pre92].

Erros em um programa podem ser considerados sob duas perspectivas: causa e efeito. Técnicas de teste têm o objetivo de revelar a existência de um erro estudando o efeito por ele produzido. Técnicas de depuração de programas têm o objetivo de estudar e eliminar a causa que o gerou. No contexto dessa disssertação o termo erro será utilizado para referenciar um defeito (causa) no programa, e o termo falha para indicar o comportamento incorreto do programa (efeito). Nesse sentido, um bom dado de teste é aquele que tem alta probabilidade de revelar um erro ainda não descoberto [Mye79].

Comumente, o teste é realizado em quatro etapas: 1) teste de unidade - teste de cada unidade implementada em seu código fonte; 2) teste de integração - teste baseado no projeto e na arquitetura do software; 3) teste de validação - validação dos requisitos de software estabelecidos na fase de análise; 4) teste de sistema - teste do software com outros elementos do sistema [Pre92].

A atividade de teste deve estar relacionada a um *Plano de Testes* que deve ser estabelecido durante a fase de planejamento de testes, onde deverão estar os objetivos do teste, recursos e estratégias a serem utilizadas, tempo disponível, etc. Com base nesse plano são executadas as fases de projeto, execução e avaliação dos casos de teste. Os casos de teste gerados são compostos pela

entrada do programa, dados de teste, mais a saída esperada.

Diferentes técnicas de teste têm sido propostas com o objetivo de selecionar bons casos de teste para detectar a maioria dos defeitos, com um custo mínimo. Dentre essas técnicas têm-se: **Técnica Baseada em Erros**: está baseada em certos tipos de erros, comumente encontrados ao longo do desenvolvimento, e utilizados para derivar requisitos de teste. Os casos de teste gerados são específicos para mostrar a presença ou ausência desses erros (ex. Análise de Mutantes [DMLS78]); **Técnica Funcional**: estabelece casos de teste baseados na especificação e na identificação dos requisitos funcionais (ex. Análise do Valor Limite [Pre92] , Grafos de Causa-Efeito [Mye79], Teste de Partição [Pre92]); **Técnica Estrutural**: utiliza o código fonte e a particular implementação para estabelecer os casos de teste (ex. Teste Baseado em Fluxo de Controle, Teste Baseado em Fluxo de Dados [RW85, Mal91]).

Essas técnicas são vistas como complementares, pois cada uma delas testa o software sob uma perspectiva diferente. Elas estabelecem um critério de teste que em geral é um dos requisitos a ser satisfeito para se considerar a atividade de teste encerrada. Satisfazer um critério significa exercitar todos os elementos requeridos por esse critério; neste caso, diz-se que o conjunto de casos de teste T utilizado é adequado em relação ao critério utilizado. Assim, um critério C é utilizado para avaliar a adequação de um conjunto de dados de teste e fornece medidas de cobertura para quantificar a atividade de teste. Além disso, um critério poderá ser utilizado para auxiliar a geração de dados de teste [RW85, FW88]. Nesse sentido, geração de dados de teste é o processo de identificar dados de entrada para o programa, tal que um critério selecionado seja satisfeito. Embora a automatização dessa tarefa seja desejável, não existe um algoritmo de propósito geral para determinar um conjunto de casos de teste que satisfaça um dado critério. Não é possível nem mesmo determinar se esse conjunto existe, ou seja, o problema de geração de dados de teste é indecidível [Cla76, Fra87].

Existem algumas limitações inerentes à atividade de teste [Mal91] que tornam a etapa de geração de dados de teste difícil e complexa de ser automatizada e a tornam um problema equivalente ao problema da parada [How75]:

- não existe um procedimento de propósito geral que possa ser utilizado para provar a corretude de um programa;
- é indecidível se dois programas computam a mesma função;
- não é sempre possível determinar se existe um dado de teste que executa uma particular seqüência de comandos do programa (caminho), ou seja, determinar se o caminho do programa é ou não executável;
- caminhos ausentes: se não existir no programa um caminho que implementa uma determinada função, não é garantido que a técnica estrutural de teste requererá que essa função seja testada e, conseqüentemente, que o erro seja revelado;

- ocorrência de correção coincidente: um item incorreto do programa é exercitado mas o programa apresenta coincidentemente um resultado correto.

A existência de caminhos não executáveis e a existência de mutantes equivalentes impedem, respectivamente, a completa satisfação dos critérios estruturais e do critério Análise de Mutantes e aumentam o esforço da atividade de teste; visto que é indecidível determiná-los, grande quantidade de esforço é gasta tentando-se gerar dados de teste para cobrir esses caminhos ou esses mutantes.

São encontradas na literatura diferentes categorias de técnicas que podem ser utilizadas para gerar dados de teste para os diversos critérios. As mais conhecidas são: geração aleatória de dados [DN84, HT88]; geração com execução simbólica [Cla76, How77, BEK75, RSBC76]; geração com execução dinâmica [Kor90]; geração de dados sensíveis a erros, cujos fundamentos permitem escolher pontos do domínio de entrada relacionados a certos tipos de erros e, por isso, com alta probabilidade de que esses sejam revelados. Nessa última categoria destacam-se as técnicas: 1) *Domain Testing* [WC80]; 2) *Constraint Based Testing* [DMO91]; 3) *(BOR/BRO) Boolean and Relational Operator Testing*; 4)*BRE(ε) Boolean and Relational Expression Testing com parâmetro* ε [Tai93]. A técnica de geração utilizada é fundamental, pois dela poderá depender a eficácia, ou seja, o número de erros revelados pelo critério.

Neste trabalho são de especial interesse os critérios de teste estrutural e as técnicas de geração de dados de teste sensíveis a erros, ou seja, com alta probabilidade de revelar erros. Na literatura, essas técnicas encontram-se dissociadas dos critérios estruturais. Os critérios estruturais requerem que certos elementos do grafo de fluxo de controle associado ao programa sejam exercitados por um conjunto de caminhos. Os critérios estruturais mais conhecidos são:

- Critérios Baseados em Fluxo de Controle: critério todos-nós, critério todos-ramos, critério todos-caminhos: exigem, respectivamente, que cada nó, que cada ramo e cada caminho do grafo seja executado pelo menos uma vez;

- Critérios Baseados em Fluxo de Dados: exigem a execução de caminhos do ponto onde uma variável foi definida, até o ponto onde ela foi utilizada (ou potencialmente utilizada) [RW85, LK83, UY88, Mal91]. Destacam-se a família de Critérios Baseados em Fluxo de Dados de Rapps e Weyuker [RW85] e a família de Critérios Potenciais Usos de Maldonado, Jino e Chaim [MCJ88, Mal91].

Estudos teóricos dos critérios estruturais têm sido conduzidos baseados principalmente na complexidade e numa relação de inclusão. A complexidade é dada pelo número de casos de teste necessários para satisfazer o critério no pior caso para qualquer programa P. Um critério C_1 inclui um critério C_2 se, para qualquer programa, todo conjunto de casos de teste T que satisfaz C_1 também satisfaz C_2. Um critério C_1 inclui estritamente um critério C_2 se C_1 inclui C_2 e C_2 não inclui C_1 [RW85, FW88]. A relação de inclusão é importante no que diz respeito ao estabelecimento de novos critérios. Ela estabelece certas propriedades dos critérios estruturais e requisitos mínimos

que eles devem preencher, tais como, incluir o critério todos-ramos e do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional [Mal91].

A aplicação dos critérios baseados em fluxo de dados sem o apoio de uma ferramenta automatizada é limitada a programas muito simples. A importância de tais ferramentas é ressaltada por vários autores [Nta88, Fra87, Wey84]. Para a família de critérios de Rapps e Weyuker destacam-se as ferramentas de teste ASSET [Fra87], PROTESTE [Sil95] e ATAC [HL90] e, para a família de critérios de Maldonado, Jino e Chaim, destaca-se a ferramenta POKE-TOOL [Cha91].

Estudos empíricos utilizando-se essas ferramentas têm sido conduzidos para verificar o comportamento dos critérios em programas reais [Wey93, Wey90, Mal91, VMJ95b]. Esses estudos têm comprovado a aplicabilidade e evidenciam o enfoque promissor dos critérios baseados em fluxo de dados. Além disso, ressaltam a grande limitação para a automatização desses critérios que é a existência de caminhos não executáveis. A maioria dos programas utilizados nesses estudos possui caminhos não executáveis.

1.2 Motivação

Dado o contexto acima, são apresentadas a seguir as principais motivações para a realização desse trabalho de pesquisa:

- A importância da atividade de teste dentro das atividades de garantia da qualidade de software;
- A necessidade de métodos e ferramentas que possibilitem a redução dos custos do teste;
- A indecidibilidade da tarefa de geração de dados de teste, que impede a sua completa automatização;
- O aspecto complementar, em termos de eficácia, das técnicas de geração de dados de teste para satisfação dos critérios;
- A grande influência da técnica de geração de dados de teste na eficácia do critério;
- A existência de poucas ferramentas de geração de dados de teste para satisfazer os diversos critérios;
- Importância de experimentos empíricos para validar as técnicas e estratégias de teste existentes;
- A comprovação obtida em diversos experimentos de que a maioria dos programas possui caminhos não executáveis;
- O grande esforço gasto em estudos empíricos para gerar dados de teste para cobrir os elementos requeridos pelos critérios e para identificar os elementos não executáveis.

1.3 Objetivos

Algumas questões podem ser colocadas quando um critério de teste estrutural é utilizado como um critério de geração de dados de teste, tais como: Quais são os melhores caminhos para se cobrir os elementos requeridos? Que pontos dentre os que executam tais caminhos devem ser selecionados? A pesquisa realizada, cujas principais contribuições são descritas nessa dissertação, focaliza técnicas de geração de dados de teste e a satisfação de critérios de teste estrutural. Tem como objetivos:

- Estudar as principais técnicas de geração de dados de teste existentes e que podem ser utilizadas com critérios estruturais;
- Propor uma estratégia de geração que aumente sobretudo a eficácia dos dados de teste gerados;
- Oferecer mecanismos de automatização da estratégia proposta que facilitem a tarefa de geração de dados de teste para cobrir os critérios estruturais;
- Reduzir os efeitos causados por caminhos não executáveis nas atividades de teste, uma das grandes limitações do teste estrutural;
- Avaliar empiricamente a validade das idéias propostas.

Em síntese, são objetivos desse trabalho: facilitar a aplicação dos critérios estruturais e ainda aumentar a probabilidade de que esses critérios possam revelar outros tipos de erros, em geral revelados por técnicas complementares; buscando reduzir os custos e aprimorando a eficácia das atividades de teste.

1.4 Organização

Neste capítulo foram apresentados o contexto no qual o trabalho se insere e as principais motivações para a sua realização e foram definidos os objetivos dessa dissertação.

No Capítulo 2 são detalhados os principais conceitos utilizados nesse trabalho. São apresentados os principais critérios de teste: funcional, estrutural e baseados em erros. São descritas as limitações da atividade de geração de dados de teste, bem como as principais técnicas existentes que podem ser utilizadas para satisfação dos critérios. Também é dada uma visão geral das estratégias de geração de dados sensíveis a erros, nas quais o trabalho está baseado.

No Capítulo 3 são definidos os Critérios Restritos, como uma forma de aumentar a eficácia dos critérios de teste estrutural. Eles combinam fundamentos das técnicas de geração apresentadas no Capítulo 2 para escolher dados que satisfazem um critério estrutural e ainda podem ser utilizados

como critérios de adequação. É descrito um estudo realizado com as técnicas de dados sensíveis a erros do Capítulo 2 e principais resultados que serviram como motivação para a introdução desses critérios. Um experimento piloto de aplicação dos Critérios Restritos é descrito e os resultados preliminares que mostram a aplicabilidade desses critérios são apresentados. Também nesse capítulo é feito um estudo das propriedades dos Critérios Restritos: complexidade e relação de inclusão.

No Capítulo 4 são propostas extensões à ferramenta POKE-TOOL com o objetivo de apoiar a utilização dos Critérios Restritos e de facilitar a etapa de geração de dados de teste. Algumas dessas extensões foram implementadas e utilizadas num experimento de aplicação dos Critérios Restritos. Esse experimento é descrito no Capítulo 5, onde são apresentados resultados obtidos comparando-se um critério restrito com o seu correspondente critério estrutural; e comparando-se uma estratégia aleatória de geração de dados de teste com uma estratégia "ad-hoc".

O Capítulo 6 sintetiza as idéias apresentadas neste trabalho, resultados observados e experiência obtida durante o desenvolvimento dessa pesquisa, e propõe estratégias de teste para a aplicação dos critérios estruturais.

O Capítulo 7 apresenta as conclusões e os desdobramentos deste trabalho.

Capítulo 2

Revisão Bibliográfica

Estratégias de geração de dados de teste visam a selecionar pontos do domínio de entrada do programa, com o objetivo de revelar erros. Técnicas de teste procuram dividir o domínio de entrada de tal maneira que os melhores pontos sejam selecionados e estabelecem um critério a ser satisfeito que poderá ser utilizado para avaliar a atividade de teste. Nesse capítulo é dada uma visão geral sobre técnicas de geração de dados de teste que visam a satisfazer os diversos critérios de teste existentes. Também são apresentados alguns aspectos da atividade de teste tais como terminologia e critérios correspondentes, e são discutidas as principais limitações dessa atividade.

2.1 Erros, Falhas e Defeitos

Engano, defeito, falha, erro, são termos bastante conhecidos da teoria de testes, no entanto, eles são utilizados de diferentes formas na literatura. Esses termos são definidos a seguir, segundo o padrão IEEE, número 610.12-1990 [IEE90]. Um engano ("mistake") é uma ação humana que produz um resultado incorreto como por exemplo uma ação incorreta tomada pelo programador. Um defeito ("bug") é um problema no programa, como por exemplo uma instrução ou comando incorreto. Uma falha ("failure") acontece quando uma saída incorreta é produzida com relação a especificação. Um erro é uma diferença entre o valor correto e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa, constitui um erro.

Classificar erros/defeitos é uma tarefa difícil e inexata. Isso, porque eles podem ser classificados considerando-se vários fatores: o comportamento do programador, o efeito causado pelo defeito no programa, etc. A maioria das classificações propostas apresenta problemas de ambigüidade, ou seja, um defeito pode ser classificado em mais de uma categoria. Existem muitos trabalhos na literatura que visam a categorizar erros/defeitos encontrados nas diferentes fases do desenvolvimento de software [GG75, How76, BP84, OW84, DMM95]. Um resumo dos diferentes trabalhos é apresentado em [VMJ95a].

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

a) Erro de domínio

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

b) Erro de computação

Figura 2.1: Tipos de Erros

Não é objetivo desse trabalho propor uma classificação para erros/defeitos. O importante é saber como os erros/defeitos se refletem nos programas, pois é de particular interesse para o teste de software que uma falha ocorra e que o erro/defeito seja revelado. Durante todo o trabalho será utilizada a classificação proposta por Howden que tem sido utilizada por vários autores [WC80, CHR82, GW86] que classifica erros no programa em:

• **Erros de Domínio**: O domínio de um caminho é dado por todas as entradas que executam o caminho dado. A cada domínio de um caminho está associada uma função. Um erro de domínio acontece quando um domínio incorreto for associado a um caminho. A Figura 2.1a apresenta um programa com um erro de domínio. O programa deve calcular a somatória $\sum_{i=1}^{n} 2^i = 2^1 + 2^2 + ... + 2^n$. A condição correta do comando *for* é $i \leq n$. Um domínio incorreto é associado ao caminho que executa esse *for*.

Erros de domínio dividem-se em dois tipos: erro na seleção de caminhos e erro do caminho ausente. No primeiro caso, um caminho incorreto é selecionado por um dado de teste porque existe um erro em uma condição em algum predicado ao longo do caminho. No segundo caso, o caminho está ausente no programa; qualquer dado de teste do domínio desse caminho executará um caminho incorreto.

• **Erros de Computação**: Um erro de computação corresponde à realização incorreta de uma computação. O exemplo da Figura 2.1b é uma outra versão incorreta para o programa e contém um erro de computação. O programa utiliza incorretamente a função *pow*, $pow(i, 2)$ eleva i ao quadrado; o desejado é elevar 2 a i. Erros de computação também se dividem em dois tipos: a computação está presente, no entanto existe um defeito e ela é computada incorretamente; ou ainda, uma computação está totalmente ausente no programa.

Erros de domínio, em geral, estão associados a defeitos em predicados do programa: defeitos em expressões booleanas (operadores e variáveis booleanas, posição e número de parênteses, etc.) e defeitos em expressões relacionais (operadores relacionais, expressões relacionais, etc.). Erros de computação estão associados a defeitos em comandos do programa que não estão relacionados com desvio de fluxo de controle. Uma computação poderá ser incorretamente realizada e causar indiretamente a avaliação incorreta de uma condição e conseqüentemente um erro de domínio. Em muitos casos, um erro em um programa pode ser visto como combinação dos dois tipos de erros apresentados.

• **Erros em Estrutura de Dados**: Um erro em estrutura de dados corresponde à definição incorreta de uma estrutura de dados. Por exemplo, a omissão de um campo em uma variável do tipo registro.

2.2 Técnicas de Teste

Existem três técnicas básicas utilizadas para derivar os casos de teste: técnica funcional, estrutural e baseada em erros. A técnica funcional deriva os casos de teste a partir das especificações do software, descritas durante as fases de análise de requisitos e/ou projeto. Exemplos de técnicas funcionais: Grafo de Causa e Efeito [Mye79], Análise do Valor Limite [Pre92], etc. A seguir as técnicas estrutural e baseada em erros são descritas com maior detalhe.

2.2.1 Técnica Estrutural

Utiliza a implementação (ou seja o código fonte do programa) para derivar os casos de teste. Geralmente, requer a execução de caminhos que exercitem determinados elementos do programa, estabelecendo-se um critério de teste estrutural. Por causa disso, esses critérios são referenciados como Critérios Baseados em Caminhos. Um exemplo de grafo de fluxo de controle pode ser visto na Figura 2.2. Um caminho é dado por uma seqüência finita de nós $n_1, ...n_i, ..n_n$ tal que n_i, n_{i+1} para $1 \leq i < n$ corresponde a uma aresta do grafo. Um caminho será completo se n_1 é o nó de entrada do grafo e n_n é o nó de saída.

A cobertura dos elementos requeridos por um critério implica na satisfação do critério dado. Medidas de cobertura do critério podem ser fornecidas e quantificam a atividade de teste. Os critérios estruturais estão baseados no grafo de fluxo de controle, na complexidade e no fluxo de dados do programa em teste. A seguir são apresentados os mais conhecidos e utilizados:

2.2.1.1 Critérios Baseados em Complexidade

• **Critério todos caminhos linearmente independentes**: [McC76, Pre92]: exige a execução

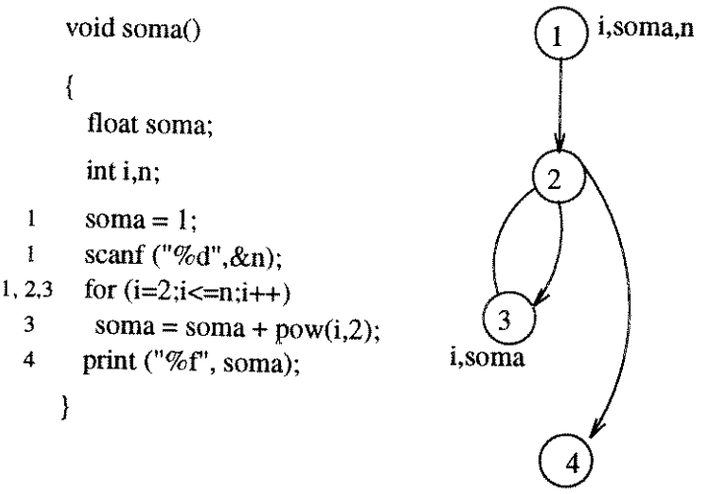


Figura 2.2: Exemplo de um Programa e o seu Grafo de Fluxo de Controle

de um conjunto de caminhos independentes do programa. Um caminho independente deve percorrer pelo menos um arco do grafo de fluxo de controle que ainda não tenha sido percorrido por nenhum outro caminho do conjunto. O número de caminhos independentes do conjunto é dado pela complexidade ciclomática do programa.

2.2.1.2 Critérios Baseados em Fluxo de Controle

Requerem a execução de componentes do grafo de fluxo de controle associado ao programa.

- **Critério todos-nós**: exige que cada nó do grafo seja executado pelo menos uma vez;
- **Critério todos-ramos**: exige que cada arco do grafo seja executado pelo menos uma vez;
- **Critério todos-caminhos**: exige que cada caminho, dado por uma seqüência finita de nós do grafo, seja executado pelo menos uma vez.
- **Critério todos-LCSAJ's**: exige a execução de todos os LCSAJ's presentes no programa. Um LCSAJ (linear code sequence and jump) [HH85] é definido como uma seqüência linear de código executável que começa ou no ponto inicial do programa ou em um ponto para o qual houve um desvio de fluxo de controle e termina com o comando final do programa ou com um comando de desvio de fluxo de controle.

Executar todos os caminhos de um programa é chamado teste ideal ou exaustivo. O teste ideal é desejável, mas na maioria das vezes impraticável. O número de caminhos de um programa pode ser muito grande e até mesmo infinito (quando laços estão presentes).

2.2.1.3 Critérios Baseados em Fluxo de Dados

Critérios de Teste Baseados em Fluxo de Dados têm o objetivo de selecionar um conjunto finito de caminhos, os mais interessantes de serem testados. Estabelecem critérios mais fortes que o critério todos-ramos e solucionam a questão da não aplicabilidade do critério todos-caminhos a qualquer programa. Exigem a execução de caminhos do ponto onde uma variável foi definida, até o ponto onde ela foi utilizada (ou potencialmente utilizada).

Uma definição de variável ocorre sempre que uma valor é armazenado na posição de memória correspondente. Um uso ocorre quando o valor de uma variável é referenciado. 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 no nós n_1 a n_m. Um caminho é simples se todos os nós exceto possivelmente o primeiro e último são distintos; é livre de laços se todos os nós são distintos. Um caminho π_1 inclui um caminho π_2 se π_2 é um sub-caminho de π_1.

Existem várias famílias de critérios baseados em fluxo de dados. Entre essas, destacam-se:

- Família de Critérios Baseados em Fluxo de Dados de Rapps e Weyuker [RW85]

Critério todos-usos: requer 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 simples 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).

- Família de critérios Potenciais Usos de Maldonado, Chaim e Jino [MCJ88]

Critério todos-potenciais-usos: requer que todas as associações potenciais-usos, do tipo (i, j, x) ou $(i, (j, k), x)$, sejam cobertas, onde o nó i possui uma definição de x e existe um caminho livre de definição c.r.a x do nó i para o nó j (ou arco (j, k)). Note que não é necessário um uso explícito de x em j (ou (j, k)), apenas que o arco (j, k) seja alcançável por um caminho livre de definição c.r.a x a partir de i, caracterizando assim um potencial-uso de x em 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_i para n_j (ou arco (n_j, n_k)), e $(n_1, ..., n_j)$ é livre de laços.

Critério todos-potenciais-usos/du: requer que todas as potenciais-associações sejam cobertas por potenciais-du-caminhos.

Um caminho completo cobre uma associação ou uma potencial-associação, se ele incluir um caminho livre de definição c.r.a x do arco i para o nó j (ou arco (j,k)). Um du-caminho ou um potencial-du-caminho é coberto por um caminho completo que o inclui.

Maldonado [Mal91] introduziu a notação $< i,(j,k),\{v_1,...,v_n\} >$ para representar o conjunto de associações $(i,(j,k),v_1),...,(i,(j,k),v_n)$ e ela indica que existe pelo menos um caminho livre de definição c.r.a todas as variáveis $v_1,..,v_n$ do nó i para o nó j (ou arco (j,k)).

As principais limitações dos critérios estruturais são: caminhos não executáveis, correção coincidente e caminhos ausentes.

Várias ferramentas foram desenvolvidas para apoiar a utilização desses critérios: a ferramenta ASSET [Fra87] e a ferramenta PROTESTE [Sil95], que apóiam os critérios de Rapps e Weyuker para programas escritos em Pascal; a ferramenta ATAC [HL90] que apóia os critérios de Rapps e Weyuker para programas escritos em C; e a ferramenta POKE-TOOL [Cha91] que apóia os critérios propostos por Maldonado et al para programas escritos em várias linguagens de programação (C, Pascal, Fortran, Cobol). A POKE-TOOL realiza a análise da adequação de um conjunto de casos de teste, e fornece medidas de cobertura dos três critérios potenciais-usos básicos apresentados acima.

2.2.2 Técnica Baseada em Erros

Os casos de teste são derivados baseando-se em defeitos específicos (ou classes de defeitos) comuns em linguagens de programação. O objetivo é mostrar a presença ou ausência de tais defeitos no programa.

- Análise de Mutantes

A idéia básica do critério Análise de Mutantes foi apresentada por De Millo [DMLS78] e é conhecida como hipótese do programador competente e assume que programadores experientes escrevem programas muito próximos do correto. Os programas incorretos contêm um desvio sintático que leva a um resultado incorreto. Geram-se programas mutantes, através de operadores de mutação, a partir do programa em teste. Os operadores são regras que serão aplicadas para definir as alterações no programa em teste. Um dado de teste deve ser gerado para diferenciar o comportamento do programa original do comportamento de um programa mutante. Quando existir essa diferença, o mutante é dito estar morto. O programa original está livre do erro descrito por esse mutante. O critério Análise de Mutantes exige que todos os mutantes sejam mortos.

A análise de Mutantes pode se tornar cara por exigir várias execuções do programa em teste e dos mutantes gerados. A principal limitação do critério é a determinação de mutantes equivalentes. Esses programas possuem sempre o mesmo comportamento que o programa original, pois eles computam a mesma função.

Um ferramenta chamada Mothra [DMGK88] foi implementada para apoiar a utilização do critério Análise de Mutantes em programas FORTRAN. A ferramenta Proteum [Del93], apóia a aplicação da análise de mutantes para programas em linguagem C, e permite a seleção de diferentes operadores de mutação, a entrada de casos de teste, e execução dos mutantes gerados; ao final, uma cobertura, representando a porcentagem de mutantes mortos, é fornecida.

A implementação de ferramentas que apoiam a utilização dos diferentes critérios de teste apresentados, possibilitou que, além dos estudos teóricos, fossem conduzidos estudos empíricos comparando o comportamente dos critérios que elas implementam (Ver Seção 2.6).

2.3 A Tarefa de Geração de Dados de Teste

Testar um programa para todos os seus possíveis valores de entrada ou executar todos os seus caminhos é idealmente desejável, mas impraticável. Os critérios descritos na seção anterior, dividem o domínio de entrada do programa em subdomínios e requerem que pelo menos um ponto de cada subdomínio seja executado. Uma vez particionado o domínio, a questão é : "Que pontos de cada subdomínio devem ser escolhidos?" Isso diz respeito à tarefa de geração de dados de teste para satisfazer um dado critério. Embora a automatização dessa tarefa seja desejável, não existe um algoritmo de propósito geral para determinar um conjunto de casos de teste que satisfaça um dado critério. Não é possível nem mesmo determinar se esse conjunto existe. Isso torna o problema de geração de dados de teste indecidível [Cla76, Fra87].

A tarefa de geração de dados de teste visa a escolher pontos do domínio para satisfazer um dado critério. Os melhores pontos são aqueles que têm maior probabilidade de revelar erros. Um gerador de dados de teste é uma ferramenta que auxilia o programador na tarefa de geração de dados de teste. Korel [Kor90] ofereceu uma classificação para os geradores de dados de teste. No trabalho apresentado nesta dissertação, é de especial interesse os geradores de dados de teste para caminhos selecionados para satisfazer um dado critério.

A seguir são discutidas, através de exemplos, as principais limitações inerentes à geração de dados de teste: correção coincidente, caminhos ausentes, caminhos não executáveis, mutantes equivalentes. Posteriormente, são apresentadas as estratégias mais utilizadas na escolha dos dados para satisfazer diferentes critérios.

2.3.1 Principais Limitações

• **Correção coincidente** : Correção coincidente ocorre quando um programa possui um defeito que é executado por um caso de teste; um estado de erro é produzido, mas coincidentemente um resultado correto é obtido. Por exemplo, considere o programa da Figura 2.1b, que apresenta o uso incorreto da função *pow*; o desejado é elevar 2 a i. Para $n = 3$, um resultado final armazenado na variável *soma* estará coincidentemente correto. O dado de teste executou incorretamente a computação, mas conicidentemente, uma saída correta foi produzida. Muitos autores [DMO91] afirmam que correção coincidente ocorre muito raramente e ao definirem suas estratégias supõem que ela não está presente.

•**Caminho Ausente**: Um exemplo de um programa para calcular a área e o tipo de um triângulo (escaleno (retângulo, obtusângulo, acutângulo), equilátero ou isóceles) é dado na Figura 2.3 [RSBC76, CR83]. Os lados devem ser fornecidos em ordem decrescente, de tal maneira que $a \geq b \geq c$. Note que os pontos dados pelo caso de teste $(a = 2, b = 1, c = 1)$ não formam um triângulo. A saída produzida deveria ser class = 0, mas entretanto não existe um caminho no programa que faça o teste $(a < b + c)$, e uma saída incorreta será produzida. Esse caminho está ausente. A técnica funcional é a mais adequada para revelar esse tipo de erro. Critérios baseados em fluxo de dados, mais especificamente os critérios Potenciais Usos, auxiliam a determinação de usos ausentes de variáveis nos programas que podem ser resultantes de computações ausentes. No entanto, critérios estruturais de teste raramente auxiliam a determinação de caminhos ausentes, isto porque eles selecionam dados de teste baseado no código do programa e a especificação não é considerada.

• **Caminhos Não Executáveis**: critérios baseados em caminhos podem exigir que caminhos não executáveis sejam exercitados; isso porque a escolha dos elementos é realizada considerando-se apenas aspectos sintáticos do programa. Um caminho é não executável se não existir um conjunto de valores para as variáveis de entrada, parâmetros e variáveis globais que causam a sua execução. O caminho 1 2 4 da Figura 2.4 é não executável pois equivale a não executar o laço, e devido à semântica do programa, isso não ocorre.

Determinar caminhos não executáveis é uma questão indecidível. O problema de caminhos não executáveis vem sendo tratado por vários autores [Fra87, MYV90, HH85, Ver92]. São três as abordagens encontradas na literatura dedicadas a esse tema: 1) determinação: destaca-se o trabalho de Frankl [Fra87] que propôs heurísticas para determinação de elementos não executáveis; 2) caracterização: visa identificar as principais causas de não executabilidade [HH85, Ver92, VMJ96a]. Quanto a previsão destacam-se os trabalhos de Malevris et al [MYV90] e Vergilio et al [VMJ93a] que estudam a influência do número de predicados na executabilidade de um caminho. Quanto maior o número de predicados de um caminho maior a probabilidade do caminho ser não executável.

```
void triangulo(a,b,c)
int a,b,c;
{  int class;   double area,as, bs, cs;
    if ((a<b) || (b<c)) {
       class = -1;      area = 0;       /* entrada nao esta forma esperada */
    }
    else if ((a!=b) && (b!=c)) {
            as = a*a;   bs = b*b;   cs = c*c;     /* escaleno */
            if (as =bs +as) {
              class = 3;   area = b*c/2.0;     /* retangulo */
            }
            else {
              s = (a+b+c)/2.0;
              area = sqrt(s*(s-a)*(s-b)*(s-c);
              if (as<bs+cs)
                  class = 4;      /* agudo */
                else
                  class = 5;      /* obtuso */
            }
        }
        else if ((a=b)&&(b=c)) {
                  class = 1;                    /* equilatero */
                  area =  a*a*sqrt(3.0)/4.0;
                }
                else {
                   class = 2;    /* isoceles */
                   if (a=b)
                      area = c*sqrt(4*a*b-c*c)/4;
                   else
                      area = a*sqrt(4*b*c-a*c)/4;
                   printf("%d", class);
                   printf("%f", area);
                }
        }
  }
```

Figura 2.3: Exemplo de Caminho Ausente

```
    void escreve()
    { int i;
1     i = 0;
2     while (i <= 100)
3     {
3       printf("%d ", i);
3       i++;
3     }
4     printf("%d ", i+100);
4   }
```

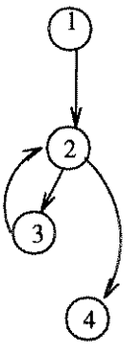


Figura 2.4: Exemplo de Caminho Não Executável

A heurística de Frankl utiliza técnicas de execução simbólica e de fluxo de dados para determinar associações não executáveis. Dada uma associação determina-se um conjunto de caminhos candidatos a cobrí-la. A idéia é eliminar os caminhos não executáveis entre os caminhos candidatos. Se o conjunto ficar vazio a associação será não executável. São analisados caminhos candidatos que passam através de laços; se o caminho que não percorre o laço for não executável e não existir caminhos livres de definição com respeito às variáveis da associação através do laço que redefinem as variáveis do predicado associado a este laço, a associação será não executável. Por exemplo, considere o programa da Figura 2.4 e a associação não executável (1,(2,4),i); não é possível executar o laço do nó 2 pois a variável *i* é redefinida por todos os caminhos através desse laço. O único caminho possível é 1 2 4 que é não executável. Módulos que utilizam essa heurística foram implementados e maiores detalhes de implementação são encontrados em [Ver92]. Uma associação não executável constitui-se um padrão de não executabilidade. Ela deverá ser removida do arquivo de associações não executadas e consequentemente todos os caminhos que a cobrem. Um padrão também será dado por uma seqüência de nós não executável. Todo caminho que incluir um padrão de não executabilidade é não executável e deve ser eliminado.

A existência de caminhos não executáveis impede muitas vezes, a completa satisfação de um critério estrutural. Um elemento requerido será não executável se não existir caminho executável que o cobre. Para tornar os critérios aplicáveis, foram derivados os critérios executáveis (ou critérios$*$) [FW88], [Mal91].

Critério todos-usos*: requer que todas as associações executáveis do programa sejam cobertas.

Critério todos-du-caminhos*: requer que todos-du-caminhos executáveis do programa sejam cobertos.

Critério todos-potenciais-usos*: requer que todas as associações potenciais-usos exe-

cutáveis do programa sejam cobertas.

Critério todos-potenciais-du-caminhos*: exige que todos-du-caminhos executáveis do programa sejam cobertos.

Critério todos-potenciais-usos/du*: requer que todas as potenciais-associações executáveis sejam cobertas por potenciais-du-caminhos executáveis.

2.3.1.1 Mutantes Equivalentes

Problema análogo ao de caminhos não executáveis é o de mutantes equivalentes para o critério Análise de Mutantes. Muitas vezes, a modificação feita no programa original para criar um mutante não altera a função implementada, ou seja o programa mutante implementa a mesma função que o programa original. Nesse caso o mutante é dito equivalente. Um programa mutante criado substituindo-se o primeiro *if* da Figura 2.3 por $if((a \leq b) + (b \leq c))$ é equivalente ao programa original. A determinação de mutantes equivalentes também é indecidível e em geral, só é possível através de heurísticas [BS79, Cra89].

2.3.2 A Geração de Dados de Teste e a Satisfação dos Critérios

A seguir são apresentadas as principais técnicas de geração de dados de teste existentes e que são geralmente utilizadas para satisfazer critérios de teste.

2.3.2.1 Geração Aleatória de Dados de Teste

A técnica de geração aleatória seleciona pontos do domínio aleatoriamente visando satisfazer um dado critério de teste. Em [BM83] uma técnica e uma ferramenta para gerar dados de teste aleatoriamente são descritas. Uma desvantagem é que em geral arquivos de dados muito grandes são gerados, e não são selecionados os "melhores" pontos. Não se garante a satisfação do critério e nenhum auxílio à determinação de elementos não executáveis é fornecido. A técnica no entanto, é defendida por vários autores [DN84, HT88] por ser menos custosa, prática e fácil de automatizar, mas não garante a cobertura dos critérios de teste.

2.3.2.2 Geração através de Execução Simbólica

A idéia de execução simbólica e sua aplicação ao teste de programas não são novas [Cla76, How77, BEK75, RSBC76]. Geralmente ela é utilizada conjuntamente com a técnica baseada em caminhos e tem o objetivo de auxiliar a geração automática de dados de teste para um dado caminho ou conjunto de caminhos. Em alguns casos é possível detectar caminhos não executáveis e além

disso, criar representações simbólicas da execução de um caminho que poderão ser checadas com representações simbólicas das saídas esperadas, facilitando a detecção de um defeito.

Técnicas de execução simbólica derivam expressões algébricas que representam a execução de um dado caminho (ou conjunto de caminhos). O resultado é uma expressão simbólica, chamada computação do caminho, que representa as variáveis de saída em termos das variáveis de entrada do programa [Fra87], e uma expressão, chamada condição do caminho, que representa condições para que o caminho seja executado, dada pelo conjunto de predicados encontrados ao longo do caminho. O domínio do caminho é dado pelo conjunto de valores que satisfazem a condição do caminho.

2.3.2.3 Técnica Dinâmica de Geração de Dados de Teste

A técnica dinâmica de geração de dados de teste foi proposta por Korel [Kor90]. Ela surgiu como uma alternativa à execução simbólica para solucionar problemas com variáveis compostas e dinâmicas. Está baseada na execução real do programa em teste, em métodos de minimização de funções e análise de fluxo de dados dinâmica. Dados reais são atribuídos às variáveis de entrada, o fluxo de execução do programa é monitorado. Se um ramo incorreto foi tomado, métodos de minimização de funções são utilizados para determinar valores para as variáveis de entrada para os quais o ramo correto seria tomado. Além disso, análise de fluxo de dados dinâmica é utilizada para determinar que variáveis de entrada são responsáveis pelo comportamento incorreto do programa. Análise dinâmica e "backtracking" permitem o tratamento de arrays e ponteiros.

2.3.2.4 Técnica de Geração de Dados Sensíveis a Erros

Técnicas de geração de dados sensíveis a erros procuram selecionar pontos do domínio que estão relacionados a certos tipos de erros e por isso têm alta probabilidade de revelar esses erros. Essas técnicas foram inicialmente definidas de uma maneira "ad-hoc". Posteriormente elas foram definidas para serem utilizadas com determinados critérios de teste. Muitas podem ser utilizadas conjuntamente com a execução simbólica, e oferecem critérios para a escolha dos valores que satisfazem a condição do caminho. Essas técnicas são apresentadas nas próximas seções.

2.4 Teste de Domínios

A técnica chamada Teste de Domínios (Domain Testing - DT) foi originalmente desenvolvida por White e Cohen [WC80], com o propósito de selecionar dados de teste para um conjunto de caminhos de programas, mas o teste de domínios não especifica como os caminhos são selecionados. Os autores sugerem que ela seja utilizada conjuntamente com Critérios de Teste Baseado em Caminhos.

O objetivo é detectar erros de domínio, selecionando dados de teste no limite do domínio do caminho ou próximos dele. Este é o fundamento da técnica de teste funcional conhecida como Análise do Valor Limite, que diz que um grande número de erros tende a se concentrar em limites dados pelas condições existentes no programa. Os limites do domínio do caminho são dados pelas condições associadas aos ramos ao longo do caminho. À cada condição corresponde um predicado que é uma combinação lógica de expressões relacionais.

White e Cohen [WC80] assumem que os predicados encontrados são simples. Interpretações de predicados em condições de caminhos determinam os limites do domínio do caminho dados por uma borda. Cada borda poderá ser aberta ou fechada, dependendo do operador relacional do predicado associado (aberta: $>, <, \neq$ e fechada: $\leq, \geq, =$). A estratégia foi definida para programas que não referenciam arrays e ponteiros e que não possuem chamadas de procedimento e/ou funções. Além disso para simplificar, as seguintes suposições foram feitas: correção coincidente não ocorre; erro do caminho ausente não ocorre para o caminho em teste; cada borda corresponde a um predicado simples (aquele que contém somente operadores relacionais); domínios adjacentes computam funções diferentes; a borda dada é linear e se estiver incorreta, a borda correta também é linear; espaço de entrada é contínuo.

A técnica de teste de domínios seleciona dois tipos de pontos de teste. Pontos *on* que pertencem a borda dada e pontos *off* que ficam a uma distância ε, muito pequena e devem pertencer ao domínio que não contém a borda. Se a borda é fechada, os pontos *on* pertencem ao domínio do caminho que está sendo testado e o ponto *off* pertence a algum domínio adjacente. Se a borda é aberta, os pontos *on* pertencem a algum domínio adjacente, enquanto os pontos *off* pertencem ao domínio em teste.

O domínio de um caminho será um poliedro convexo de dimensão N, para programas com interpretações de predicados que resultam em inequações simples e lineares com relação às variáveis de entrada do programa, onde N é o número de entradas do programa. A estratégia NX1 propõe que sejam escolhidos N pontos *on* e um ponto *off* para cada borda do domínio do caminho. Por que a estratégia seleciona N pontos *on* e um ponto *off* ela é chamada de estratégia NX1.

A Figura 2.5 apresenta o domínio de entrada para o caminho 1 2 3 4 3 5 6, aplicando-se a estratégia 2X1 (teste em duas dimensões) na borda que corresponde ao último *if* do programa, P = (0,1.001) e Q = (1,1.001) poderiam ser escolhidos como pontos *on* e V = (0.5, 1.001+ε), onde ε é o menor valor possível, como ponto *off*. Os pontos *on* devem ser escolhidos tão próximos quanto possíveis do final da borda.

Alguns problemas com a técnica Domain Testing básica são descritos na literatura e algumas soluções são propostas [CHR82] [ZAW92].

```
     ....
1    leitura(x,y);
1    if (x>= 0)
2    {
2     z =y-x;
2     d = 0;
3     while (z>d)
4       d = d+1;
5     if (x<=d)
6       if (y <= 1.001*d)
     ....
```

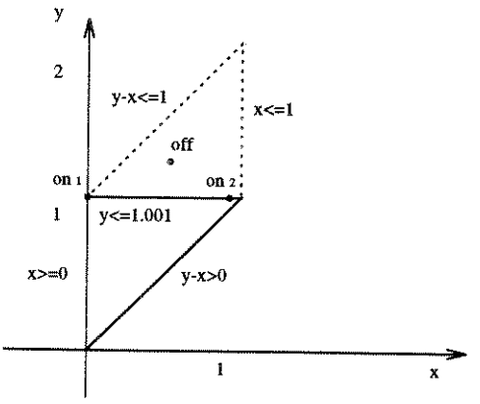


Figura 2.5: Ilustrando a Estratégia 2X1 - Domain Testing

2.5 Teste Baseado em Restrições

A técnica de teste baseado em restrições (Constraint-Based Testing - CBT), foi proposta por DeMillo e Offut em 1991 [DMO91]. A técnica tem como objetivo auxiliar a geração de dados de teste adequados ao teste de mutação. Ela usa restrições algébricas projetadas para matar mutantes, ou seja para detectar defeitos descritos pelos operadores de mutação. Para matar um mutante é necessário que 1) o comando mudado seja executado (condição de alcançabilidade); 2) nesse ponto, após a execução do comando mudado, o estado do programa original seja diferente do estado do programa mutante (condição de necessidade); 3) essa diferença seja propagada até que comandos de saída sejam executados e resultados diferentes sejam produzidos, possibilitanto assim a revelação do defeito (condição de suficiência).

Derivar dados de teste que satisfaçam a condição de suficiência é impraticável [DMO91, How76, BA82, Mor90, RT88]. Budd e Angluim [BA82] relacionam o problema ao termo correção coincidente, sendo esta uma questão indecidível. A Figura 2.6 apresenta um programa para calcular o máximo de dois números M e N, ele possui 4 comandos numerados a esquerda. Foram criados 2 mutantes do programa original correspondentes às duas mutações especificadas a esquerda pela letra grega Δ. Para se alcançar o comando de número 1 não é necessário nenhuma restrição. Para se alcançar o de número 3 é necessário $(N > M)$. As condições de necessidade para que estados diferentes sejam produzidos após a execução dos comandos 1 e 3, são respectivamente $M \neq N$ e $N \neq 6$.

Em experimentos conduzidos por De Millo et al [DMO91], notou-se que correção coincidente raramente acontece na prática. A probabilidade de se resolver a condição de suficiência dado que as condições de necessidade e alcançabilidade foram satisfeitas é muito grande. Em 90% dos mutantes analisados, (quando predicados não estavam envolvidos), a produção de um estado

```
   void max()
   { double max,m,n;
1    max = m;
Δ    max = n;
2    if (n>m)
3      max = n;
Δ      max = 6;
4    return max;
   }
```

Figura 2.6: Exemplificando as Restrições de Alcançabilidade e Necessidade

Programa Original	if ((i+k)>=j)
Mutante	if ((3+k)>=j)
Restrição de Necessidade	i <> 3
Caso de Teste	I = 7, J = 9, K = 7

Figura 2.7: Problema com Predicados

intermediário incorreto, garantiu um estado final incorreto e conseqüentemente a morte do mutante. A posição adotada por DeMilo et al foi então assumir que a satisfação das condições de necessidade e alcançabilidade implica na satisfação da condição de suficiência.

Por outro lado, apenas 60% dos mutantes que envolviam mutações em predicados foram mortos. Em muitos casos, embora um efeito no estado do mutante fosse registrado, o resultado da avaliação de um predicado continuava o mesmo que o do programa original. Um exemplo desse problema é apresentado na Figura 2.7. A condição de necessidade gerada para provocar a diferença de estado é $I \neq 3$. O caso de teste $I = 7, J = 9, K = 7$ satisfaz essa restrição, mas o predicado continua sendo avaliado true, como era esperado.

Para o problema de suficiência, De Millo propôs restrições de predicados, que são extensões das restrições de necessidade e asseguram diferenças no resultado da avaliação de predicados. Para o exemplo acima a restrição de predicado é dada por $(I + K > J) \neq (3 + K > J)$. O caso de teste $I = 7, J = 9, K = 7$, não satisfaz essa restrição, $(7 + 7 > 9) \neq (3 + 7 > 9) = False$. Já o caso de teste $I = 7, J = 10, K = 7$, provocará diferença no resultado da avaliação, $(7 + 7 > 10) \neq (3 + 7 > 10) : True \neq False = True$ e produzirá a avaliação incorreta do predicado.

Um conjunto de ferramentas chamado Godzila foi implementado com o objetivo de gerar dados de teste automaticamente. Ele gera e resolve restrições para detectar erros descritos pelos mutantes gerados pelo sistema Mothra [DMGK88]. No Apêndice A encontram-se as tabelas geradas

por DeMilo [DMO91] que mostram as restrições utilizadas pela ferramenta Godzila para cada operador de mutação da ferramenta Mothra.

2.6 Teste Baseado em Predicados

Técnicas de teste referenciadas como teste baseado em predicados requerem em geral teste para cada predicado (ou condição) do programa. As técnicas baseadas em predicados foram classificadas por Tai [Tai93] e se dividem em dois grupos. As que testam predicados simples e as que testam predicados compostos. O problema com predicados compostos é que nem sempre é possível testar todas as combinações possíveis e não adianta testar todas as possíveis situações individuais para todos os predicados. Poderá acontecer que embora os predicados individuais sejam avaliados incorretamente, o resultado da avaliação de toda a expressão poderá ser coincidentemente correto. Para o predicado composto $P = (E_1 = E_2)\&(E_3 < E_4)$, os seguintes testes são requeridos.

$t_1 : E_1 = E_2 e E_3 = E_4$; $\quad t_2 : E_1 > E_2 e E_3 > E_4 \quad t_3 : E_1 < E_2 e E_3 < E_4$

Note que para esse conjunto de testes todas as possíveis situações para os dois predicados simples foram consideradas. Mas entretanto, o conjunto não diferencia o predicado P do predicado $R = (E_1 \neq E_2)\&(E_3 = E_4)$

Essa situação é a mesma observada por DeMillo [DMO91] no teste baseado em restrições, o qual sugeriu o uso de uma restrição adicional chamada restrição de predicado, que é uma aproximação da condição de suficiência.

Tai [Tai93] propõe duas técnicas de teste para predicados compostos. As técnicas são baseadas em erros (análogos à mutação dos operadores relacionais e lógicos). Pois, requerem que dados de teste sejam executados para satisfazer um conjunto de restrições. O conjunto de restrições é projetado para garantir que defeitos em operadores booleanos e defeitos em operadores relacionais sejam detectados (com a suposição de que não existam defeitos de outros tipos e nem restrições não executáveis).

A primeira técnica chama-se Boolean Operator Testing (ou BOR), garante a detecção de defeitos em operadores booleanos. Tai afirma que se uma técnica é efetiva em determinar defeitos em operadores booleanos, ela também o será em determinar defeitos em expressões booleanas. A segunda chama-se Boolean and Relational Operator Testing (ou BRO) e garante a detecção de defeitos em operadores booleanos e relacionais. Tai utiliza a seguinte notação para definir as restrições que serão requeridas pelas técnicas BRO e BOR.

Para uma variável booleana B:

- t: denota que B vale true;

- f: denota que B vale false;

Para uma expressão relacional $(E_1 rop E_2)$

- t: denota valor true para a expressão;
- f: denota valor false para a expressão;
- $>$: denota que $(E_1 - E_2) > 0$
- $<$: denota que $(E_1 - E_2) < 0$
- $=$: denota que $(E_1 - E_2) = 0$
- $+\varepsilon$: denota que $0 < (E_1 - E_2) \leq \varepsilon$
- $-\varepsilon$: denota que $-\varepsilon \leq (E_1 - E_2) < 0$

onde ε é um número pequeno.

Dado o predicado $C = (E_1 \geq E_2)|\neg(E_3 > E_4)$, uma restrição para C é dada por uma lista denotando os valores citados acima, para as variáveis booleanas ou expressões relacionais. A restrição dada pela lista $X = (=, <)$ requer um teste fazendo $E_1 = E_2$ e $E_3 < E_4$. Note que o operador "$\neg$ " não afeta o requisito $<$. O valor produzido por C quando satisfeita a restrição X é dado por C(X). O conjunto de todas as restrições para um predicado C é dado por S que é dividido em dois conjuntos. O primeiro conjunto $S_t(C)$ composto por todas as restrições X, tais que C(X) = true; e o segundo conjunto $S_f(C)$ composto por todas as restrições X, tais que C(X) = false. A concatenação de duas restrições (listas) l_1 e l_2 é denotada por (l_1, l_2).

Sejam A e B dois conjuntos de restrições. $A\%B$ denota o conjunto mínimo de elementos (u, v), tais que $u \in A$ e $v \in B$ e cada elemento de A e de B aparece como u ou v pelo menos uma vez. Se $A = \{(=), (>)\}$ e $B = \{(<), (>)\}$, $A\%B$ tem dois valores possíveis: $\{(=, <), (>, <)\}$ ou $\{(=, >), (>, <)\}$. Se $A = \{(=), (>)\}$ e $B = \{(<), (>), (=)\}$, $A\%B$ terá seis valores possíveis: $\{(=, <), (>, >), (=, =)\}, \{(=, <), (>, >), (>, =)\}, \{(=, <), (=, >), (>, =)\}, \{(>, <), (=, >), (>, =)\}, \{(>, <), (=, >), (=, =)\}$ e $\{(>, <), (>, >), (=, =)\}$. Ainda definem-se $A\$B$ como união de A e B e $A * B$ como produto de A e B.

As técnicas BOR e BRO exigem as seguintes restrições:

- Para uma variável booleana B: {(t),(f)}
- Para uma expressão relacional $E_1 rop E_2$

  BOR: {(t),(f)},

  BRO: $\{(>), (=), (<)\}$

- para uma expressão booleana $C_1 bop C_2$, sendo S_1 e S_2 conjunto de restrições para os predicados C_1 e C_2 respectivamente.

  1. $C_1 or C_2$.

     $F(C) = S_{1f} \% S_{2f}$

     $T(C) = \{S_{1t} * \{f_2\}\} \$ \{\{f_1\} * S_{2t}\}$

     onde $f_1 \in S_{1f} e f_2 \in S_{2f} e (f_1, f_2) \in F(C)$

     BOR = BRO = T(C) $ F(C)

  2. $C_1 and C_2$.

     $T(C) = S_{1t} \% S_{2t}$

     $F(C) = \{S_{1f} * \{t_2\}\} \$ \{\{t_1\} * S_{2f}\}$

     onde $t_1 \in S_{1t} e t_2 \in S_{2t} e (t_1, t_2) \in T(C)$

     BOR = BRO = T(C) $ F(C)

Para o predicado $P = ((E_1 < E_2) \& ((E_3 \geq E_4) | (E_5 = E_6)))$, a construção das restrições é feita de uma forma botom-up. Seguindo as técnicas BRO e BOR definidas anteriormente, geram-se conjunto de restrições S_1, S_2 e S_3 correspondente aos predicados C_1, C_2 e C_3. Gera-se um conjunto S_4 para a expressão $C_1 or C_3$ utilizando-se S_2 e S_3. Gera-se um conjunto S_5, a partir de $S_1 e S_4$ para a expressão $C_1 and (C_2 or C_3)$. A Figura 2.8 mostra o total de restrições requeridas.

Existe uma ferramenta, chamada BGG, proposta por Tai para gerar conjunto de restrições BRO para predicados de programas escritos em Pascal. Ela fornece a cobertura do conjunto de restrições gerado para um dado conjunto de testes. As restrições geradas pelas técnicas BRO e BOR podem ser utilizadas para gerar dados de teste adequados à análise de mutantes e também ser utilizadas tanto com técnicas de teste funcional como com técnicas estruturais. Na prática, uma restrição também poderá ser não executável, caso no qual nem todas as restrições derivadas poderão ser satisfeitas e conseqüentemente não se poderá garantir que todos os erros de predicados foram detectados.

Com o objetivo de aumentar a eficácia das técnicas, Tai [Tai93] propôs uma extensão chamada Boolean and Relational Expression Testing com parâmetro ε (or BRE(ε) testing). A idéia é substituir ocorrências de $>$ e $<$ por $+\varepsilon$ e $-\varepsilon$, respectivamente, onde ε é um número muito pequeno, maior que 0. Esta técnica usa fundamentos da técnica Domain Testing e é mais eficaz para determinar defeitos em expressões relacionais envolvendo outros tipos de erros além dos detectados pelas técnicas BOR/BRO.

2.7 Estudos Teóricos e Empíricos de Critérios de Teste

Estudos teóricos e empíricos permitem a comparação dos diversos critérios de teste existentes e têm sido conduzidos com o objetivo de se encontrar formas econômicas e produtivas para

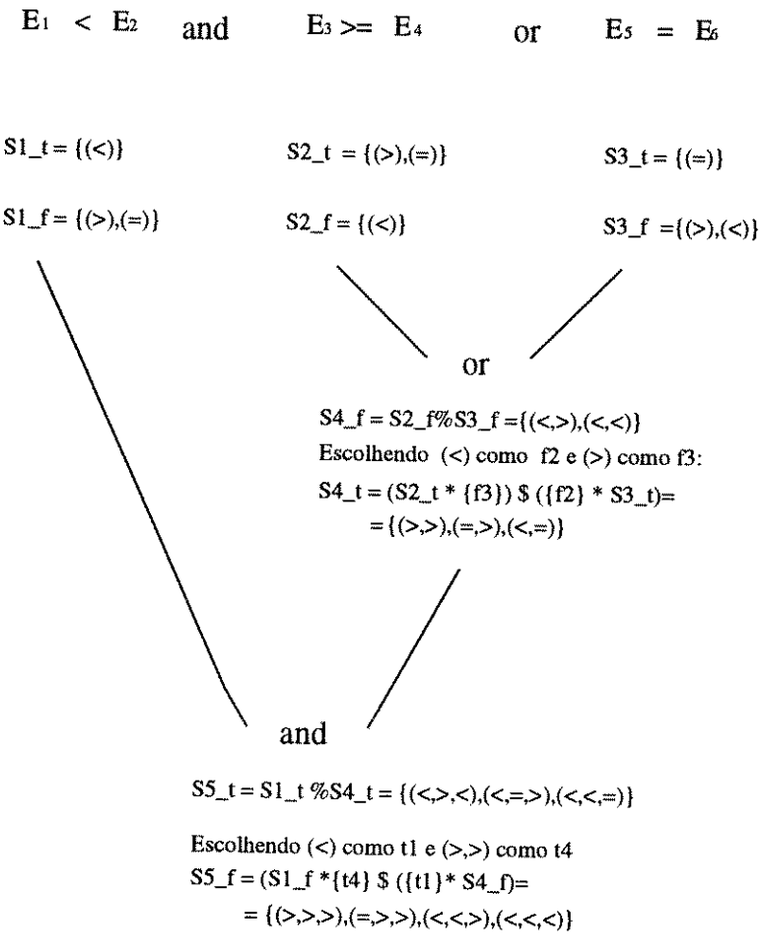


Figura 2.8: Restrições BOR/BRO Requeridas - Exemplo

se produzir testes. Segundo Wong [Won93] custo, eficácia e dificuldade de satisfação ("strength") são os fatores básicos para comparar a adequação dos critérios de teste: 1) Custo: refere-se ao esforço necessário para utilizar o critério e é geralmente medido pelo número de casos de teste necessários para satisfazer o critério dado; 2) Eficácia: refere-se à capacidade de um critério em revelar um número maior de erros em relação a outro; 3) Dificuldade de Satisfação: refere-se à probabilidade de satisfazer um critério tendo satisfeito um outro. Uma visão geral dos estudos realizados, considerando esses fatores é dada a seguir.

2.7.1 Comparação entre os Critérios Estruturais

Comparações entre os critérios estruturais de teste têm sido realizadas baseadas principalmente numa hierarquia, dada por uma relação de inclusão e no cálculo da complexidade do critério (número de dados de teste necessários no pior caso).

Um critério C_1 inclui um critério C_2 se todo e qualquer conjunto de caminhos que satisfaz C_1, ou seja, que cobre os elementos requeridos por C_1, ou ainda é C_1-adequado, também satisfaz C_2. Um critério C_1 inclui estritamente um critério C_2 se C_1 inclui C_2 e C_2 não inclui C_1 [FW88]. Dois critérios são incomparáveis se C_1 não incluir C_2 e nem C_2 incluir C_1. Um critério C_1 inclui estritamente um critério C_2 se C_1 inclui C_2 mas C_2 não inclui C_1.

A relação de inclusão entre os critérios apresentados é dada na Figura 2.9a [Mal91, FW88]. Note que os critérios baseados em fluxo de dados são mais exigentes que os baseados em fluxo de controle e fazem a ponte entre o critério todos-ramos e todos-caminhos, além disso, os critérios Potenciais Usos são os mais exigentes e incluem os demais.

A presença de caminhos não executáveis altera a relação de inclusão entre os critérios. Os critérios de Rapps e Weyuker deixam de incluir o critério todos-ramos [Mal91, FW88]. A nova relação é dada na Figura 2.9b. Note que os critérios Potenciais-Usos continuam incluindo o critérios todos-ramos mesmo na presença de caminhos não executáveis.

A relação de inclusão é importante no que diz respeito ao estabelecimento de novos critérios. Ela estabelece certas propriedades dos critérios estruturais e requisitos mínimos que eles devem preencher, tais como, incluir o critério todos-ramos e do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional [Mal91].

A relação de inclusão tem sido bastante utilizada para comparar critérios quando considerados os fatores 1 e 3 (custo e eficácia). Por outro lado, muitos autores não consideram a relação de inclusão um meio para avaliar a eficácia dos critérios (fator 2). Frankl and Weyuker [FW93b, FW93a] exploram a relação de inclusão entre critérios e diz que o fato de C_1 incluir C_2 não garante que C_2 seja melhor para revelar defeitos que C_1. Segundo Weyuker et al [WJ91], a relação de inclusão é útil para comparar o custo entre os critérios, mas não diz nada sobre a eficácia destes em revelar defeitos. Hamlet [HT88] diz que é possível um critério C_1 incluir um critério C_2, e que

um conjunto de casos de teste T_2 (que satisfaz C_2) revele defeitos não revelados por um conjunto T_1 (que satisfaz C_1). Isso porque a eficácia de T_1 e de T_2 dependerá da estratégia utilizada para gerá-los.

Weyuker [Wey93, Wey90] apresenta um estudo para determinar uma maneira de estimar o número de casos de teste necessários para satisfazer um dado critério, para um programa P. O estudo empírico indicou que o número de casos de teste requeridos para satisfazer os seus critérios pode ser visto como linear em função do número de comandos de decisão do programa. Outro ponto importante é a caracterização do pior caso empírico, denominado complexidade empírica: o máximo valor obtido dividindo-se o número de casos de teste pelo número de comandos de decisão. Até mesmo nesses casos, os estudos mostram que os critérios baseados em fluxo de dados são aplicáveis.

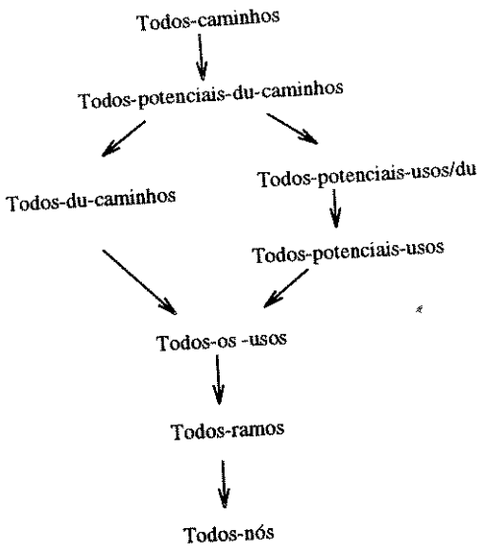
Os critérios Potenciais Usos também foram utilizados num experimento de aplicação dos mesmos programas utilizados por Weyuker. Vários modelos foram obtidos relacionando o número de casos de teste e várias características de programa (tais como número de definições, número de variáveis, número de nós, complexidade, etc). Na prática, o número de casos de teste demandados foi pequeno [Mal91].

Um outro estudo comparativo de critérios foi realizado utilizando a ferramenta ATAC e a POKE-TOOL [VMJ95b] . Na prática, o modelo de instrumentação das ferramentas afetou a relação de inclusão. Percebeu-se também a influência da linguagem C. A análise de inclusão foi conduzida por Frankl e Weyuker considerando algumas supossições, entre elas: que cada predicado e cada comando deve fazer referência a pelo menos uma variável. Mas como relatado em [HL90], muitos programas escritos em linguagem C não possuem essas características. Por exemplo o trecho de programa da Figura 2.10. Nesse programa não existe p-uso associado ao arco (v,y) e nem c-uso associado ao nó y. Portanto, todos-usos não inclui todos-ramos (Figura 2.11). Já os Critérios Potenciais Usos exigem apenas a definição de pelo menos uma variável no nó de entrada do programa para continuar incluindo o critério todos-ramos [Mal91]. Isso é satisfeito pela grande maioria dos programas.

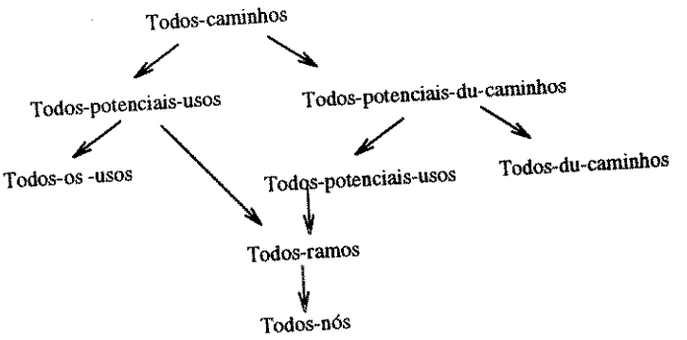
2.7.2 Comparação Entre Diferentes Tipos de Mutação

Um problema com a análise de mutantes é o seu alto custo. Isso ocorre porque um alto número de mutantes é gerado e precisam ser executados e analisados pelo testador demandando um grande esforço. Várias estratégias visando diminuir esse esforço têm sido propostas e analisadas empiricamente pela relação custo/eficácia.

Mathur e Wong [MW93] compararam duas formas alternativas para reduzir o custo necessários com a Análise de Mutantes: a Mutação Aleatória e a Mutação restrita. Na Mutação aleatória, 10% do número de mutantes possíveis são gerados. Na Mutação Restrita, alguns opera-



a) Ordem Parcial entre Critérios



b) Ordem Parcial entre Critérios

na Presença de Caminhos não Executáveis

Figura 2.9: Relação de Inclusão entre os Critérios Estruturais

```
.....
v  if (getchar() != EOF)  A
x     char_ent++;         B
      else
y     exit();             C
z                 ....
```

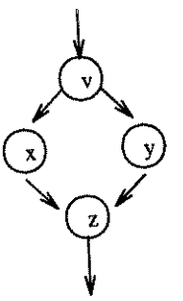


Figura 2.10: Trecho de Programa em Linguagem C que Altera Relação de Inclusão

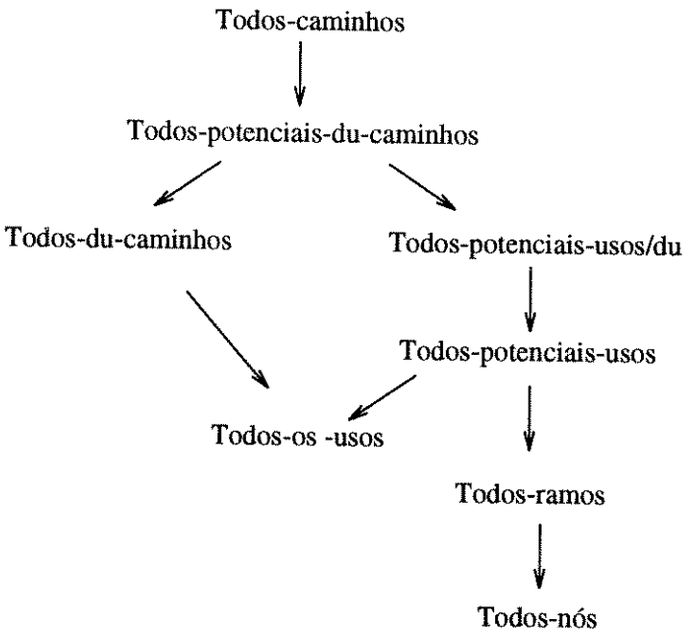


Figura 2.11: Nova Ordem Parcial entre os Critérios Considerando a Linguagem C

dores de mutação específicos são escolhidos. Segundo os autores, ambas as formas apresentaram-se eficientes, houve uma redução do número de mutantes sem perda significativa de eficiência em revelar os erros.

Nos trabalhos de Souza [Sou96], Wong et al [WMM94] e Offut et al [Oea96] foram comparadas 6 classes diferentes de mutação restrita quanto à eficácia. Deste experimentos pôde-se observar que é possível reduzir o número de mutantes. Foram determinadas classes de mutação mais econômicas (baixo custo de aplicação) e eficazes e foi estabelecida uma ordem incremental para o emprego dessas classes de mutação [Sou96].

2.7.3 Comparação Entre Critérios Baseados em Fluxo de Dados e Análise de Mutantes

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 [WMM94], Mathur et al [MW93], Offut [Oea97] realizaram estudos comparando o critério Análise de Mutantes com o critério baseado em fluxo de dados Todos-Usos, e obtiveram resultados semelhantes. Na prática, o critério Análise de Mutantes demandou um número maior de casos de teste e se mostrou mais eficiente. Além disso, os conjuntos de teste adequados ao critério Análise de Mutantes foram todos-usos adequados, não sendo o inverso verdadeiro.

Souza et al [Sou96], [SMV97] apresentam resultados de um estudo empírico com a finalidade

Tabela 2.1: Quadro comparativo das Técnicas de Geração

Técnica	Critério Associado	Se aplica Pred. Compostos	Tipos de Erros	Ferramentas Disponíveis	Resolver Restrições
DT	Baseado em Caminhos	não	dom.		sim
CBT	Análise de Mutantes	sim	comp. e dom.	Godzila	sim
BOR/BRO	–	sim	dom.	BGG	sim
BRE	–	sim	dom.		sim

de avaliar o "strenght" e o custo dos critérios Análise de Mutantes e Potenciais Usos. O custo obtido para a Análise de Mutantes, dado pelo número de casos de teste, foi maior. Com relação ao "strenght", dificuldade de satisfação, os dois critérios se mostraram incomparáveis.

2.8 Considerações Finais

Nesse capítulo foram apresentados aspectos da atividade de teste, particularmente da etapa de geração de dados de teste que têm o objetivo de satisfazer critérios de teste existentes e de revelar o maior número de erros possíveis. Foram apresentadas a terminologia da área e uma visão geral dos principais trabalhos que oferecem uma classificação para defeitos/erros. Foram discutidas, através de exemplos, as principais limitações das atividades de teste: correção coincidente, caminhos ausentes, caminhos não executáveis, mutantes equivalentes, mostrando que a atividade de geração de dados de teste é bastante dificultada por essas limitações.

São quatro as principais técnicas apresentadas para realizar a geração de dados de teste para satisfazer os critérios.

- **Geração aleatória**: gerar dados de teste aleatoriamente não garante a seleção dos melhores pontos, também não auxilia a determinação de elementos não executáveis. A técnica, no entanto, é defendida por vários autores [DN84], [HT88] por ser menos custosa, prática e fácil de automatizar.

- **Geração com execução simbólica**: apesar de bastante difundida, apresenta muitas dificuldades de automatização. Ela é mais custosa que a aleatória, mas foi comprovada ser mais eficaz em vários experimentos. Permite em alguns casos detectar caminhos não executáveis e além disso, criar representações simbólicas da execução de um caminho que poderão ser checadas com representações simbólicas das saídas esperadas, facilitando a detecção de um defeito. A execução simbólica tem sido utilizada também em outras áreas da computação, tais como depuração [CR83].

- **Geração com execução dinâmica**: visa resolver os problemas da execução simbólica. Existem poucos experimentos realizados com execução dinâmica. O ideal é utilizar conjuntamente

execução simbólica com execução dinâmica.

• **Geração de dados sensíveis a erros**: não oferecem uma maneira automática de gerar o dado, mas são interessantes por gerarem dados de teste com alta probabilidade de revelar erros e devem ser utilizadas com execução simbólica e/ou dinâmica.

Não existem na literatura estudos teóricos e/ou empíricos que visam a comparar as diferentes técnicas de geração de dados de teste sensíveis a erros. A Tabela 2.1 resume as principais características de cada técnica apresentada: o critério de teste para o qual foi proposta; se ela se aplica a predicados compostos; a que tipos de erros está associada; se existe alguma ferramenta disponível; e se existe necessidade de se resolver restrições.

A técnica Domain Testing deve ser utilizada com um critério de seleção de caminhos. Se um critério mais exigente for utilizado, será maior a probabilidade de um caminho que revela o erro ser selecionado. Essa técnica também é a mais difícil de ser aplicada e automatizada, existem problemas de aplicação com predicados compostos e não lineares, com domínios discretos, etc.

A técnica Constraint Based Testing tem sua eficácia dependente dos operadores de mutação utilizados, visto que não é possível descrever todo tipo de erro. As restrições de predicado propostas são muito importantes pois aproximam-se das restrições de suficiência.

As técnicas BOR/BRO são mais práticas e mais fáceis de serem automatizadas, mas são menos poderosas em termos de eficácia. BRE parece ser mais eficaz, mas por outro lado pode existir dificuldade em se determinar o valor de ϵ em domínios discretos.

Um problema inerente à todas técnicas de geração de dados sensíveis a erros é que no final, um conjunto de restrições é derivado e são necessários programas que resolvam eficientemente essas restrições, através de execução simbólica ou dinâmica. A indecidibilidade da atividade de geração de dados de teste permanece, não sendo possível sua completa automatização.

Devido a inexistência de estudos empíricos com as técnicas existentes, foi realizado um experimento de aplicação das mesmas a um conjunto de programas simples e incorretos, que são comumente utilizados na literatura. O objetivo desse experimento que será descrito no Capítulo 3, foi verificar a dificuldade de aplicação, a eficácia, e o número de casos de teste necessários para satisfazer diferentes critérios, para cada técnica. A partir desse experimento foi possível:

• Estudar os diferentes conceitos e aspectos complementares associados a cada técnica, e a utilização desses para aumentar a eficácia dos critérios estruturais de teste (Capítulos 3 e 5).

• Facilitar a automatização da geração de dados de teste para satisfazer os critérios, estabelecendo um conjunto de restrições a serem resolvidas (Capítulo 4) e uma extensão à ferramenta POKE-TOOL.

• Estabelecer uma estratégia e um conjunto de diretrizes para auxiliar a atividade de teste (Capítulo 6).

Capítulo 3

Definição de Critérios de Teste Estrutural Baseados em Restrições

Foi visto no capítulo anterior, que os critérios de teste estrutural utilizam informação sobre o código do programa em teste para derivar os dados de teste. Geralmente, o domínio de entrada é dividido em sub-domínios, e um ponto de cada sub-domínio deve ser selecionado. Mas como realizar a escolha desse ponto? Técnicas de geração de dados de teste sensíveis a erros possuem fundamentos que permitem a escolha de pontos que descrevem possíveis erros no programa. Nesse capítulo são introduzidas extensões para as diversas famílias de critérios de teste estrutural. Essas extensões utilizam os princípios das técnicas de geração de dados sensíveis a erros para aumentar a eficácia desses critérios, ou seja, aumentar a capacidade destes de revelar erros, auxiliando a escolha dos pontos em cada sub-domínio.

A proposição dos Critérios Restritos foi motivada por um estudo empírico onde as técnicas de geração de dados de teste sensíveis a erros descritas no Capítulo 2 foram aplicadas. Primeiramente, são apresentados uma síntese desse estudo e principais resultados obtidos, além de um exemplo mostrando um caso onde a combinação dessas técnicas é vantajosa. Posteriormente, os critérios baseados em restrições são definidos e os resultados de um experimento piloto de aplicação são apresentados. Ao final, são discutidas as suas principais propriedades: aspectos de complexidade e de inclusão entre eles, e os demais critérios.

3.1 Motivação

3.1.1 Aplicando as Técnicas de Geração de Dados de Teste

As técnicas de geração de dados de teste, vistas no Capítulo 2, têm como objetivo selecionar bons dados de teste, ou seja, dados de teste com alta probabilidade de revelar erros. Foi realizada uma avaliação empírica de tais técnicas utilizando-se um conjunto de 6 programas, extraídos da

Tabela 3.1: Descrição dos Comandos Incorretos em Cada Versão

Programa	Comando Correto	Comando Incorreto
max-1	$max = m$	$max = abs(m)$
max-2	$if(n >= m)$	$if(n <= m)$
max-3	$max = m$	$max = n$
whi80-1	$if(k >= (i+2))$	$if(k >= (i+1))$
change-1	$j = j + l$	$j = j + 1$
sum-1	$for(i = 1; i <= n; i++)$	$for(i = 1; i < n; i++)$
sum-2	$s = s + pow(i, 2)$	$s = s + pow(2, i)$
triangle-1	$if(c <= 0 \|\| a > (b + c))$	$if(c <= 0)$
voas-1	$d = b * b - 4 * a * c$	$d = b * b - 5 * a * c$

literatura: *max* [DMO91], *whi80* [WC80], *change* [VMJ93b], *sum* [VMJ95a], *triangle* [RSBC76] e *voas* [VMM91], cujos erros estão relatados nas respectivas referências citadas e estão sintetizadas na Tabela 3.1.

Cada erro foi colocado em um programa correto por vez. As técnicas foram aplicadas às diferentes versões incorretas de um mesmo programa. Por exemplo, para o programa *max*, existem três versões incorretas, *max-1*, *max-2*, *max-3*, cada uma com um único erro associado. Os erros foram classificados segundo as categorais descritas no capítulo anterior, em erros de domínio e erros de computação. Foram determinadas as condições necessárias e suficientes para a revelação de cada erro. Esses dados são apresentados na Tabela 3.2. Os dois últimos programas dessa tabela são especiais. O programa *triangle-1* tem um caminho ausente e o programa *voas-1* apresenta correção coincidente em muitos casos.

A aplicação das quatro técnicas Domain Testing - DT, Constraint Based Testing - CBT, Boolean and Relational Operator Testing - BRO, e Boolean and Relational Expression Testing with parameter ε - BRE(ε) foi realizada manualmente. A técnica DT foi aplicada ao conjunto de caminhos escolhidos para satisfazer o critério todos-potenciais-usos implementados pela ferramenta POKE-TOOL. Exceto para o programa *sum*, que contém laços, sempre foi necessário executar todos os caminhos do programa para satisfazer o critério, portanto a análise da eficácia da técnica DT não foi influenciada pelo critério estrutural escolhido.

Um problema com o teste de domínios diz respeito a predicados compostos e a predicados não lineares, que é o caso dos dois últimos programas da Tabela 3.2. Nesses casos a técnica DT não foi aplicada, pois isso exigiria um estudo das sugestões propostas em [ZAW92] e uma extensão da técnica para lidar com essas situações. Também foram necessárias na maioria dos programas, algumas adaptações da técnica DT para lidar com domínios ilimitados e discretos. Foram limitados os domínios dos programas *max*, *whi80* e *change*. Problemas com domínios discretos foram contornados visto que a aplicação da técnica foi manual. Uma dificuldade foi determinar a menor

distância ε da borda dada, dificuldade encontrada também durante a aplicação da técnica BRE(ε).

A técnica CBT foi aplicada utilizando-se a ferramenta Proteum. Foi derivado um conjunto de condições necessárias e suficientes para a morte dos mutantes gerados pelos operadores da Proteum. Esse conjunto é apresentado no Apêndice A e baseia-se fortemente nas restrições propostas por DeMillo para o sistema Mothra-Godzila do Capítulo 2. Um erro foi considerado ser sempre revelado pela técnica CBT, quando foi possível encontrar nesse conjunto de condições, a condição de necessidade e suficiência para a revelação do erro considerado.

O experimento consistiu basicamente de dois passos:

- Análise de eficácia: foi realizado um estudo da capacidade da técnica de revelar o erro em questão. Para esse estudo três possíveis respostas podem ser encontradas:

  1. A técnica nunca revela o erro: não existe um conjunto de dados de teste gerado pela técnica capaz de revelar o erro.
  2. A técnica pode revelar o erro: existe ao menos um conjunto de dados de teste gerado pela técnica que revela o erro e existe pelo menos um conjunto de dados de teste que não é capaz de revelar o erro.
  3. A técnica sempre revela o erro: todos os conjuntos gerados pela estratégia revelam o erro.

  Note na Tabela 3.2 que não foi obtida resposta número 1 em nenhum caso.

- Geração manual de dados de teste: conjuntos de casos de teste foram derivados para cada programa e o número de casos de teste necessários foi anotado para cada técnica. Em nenhum momento procurou-se minimizar esse número. Quando uma resposta 3 foi obtida, apenas um conjunto (aquele que revela o erro) foi derivado. Se uma resposta 2 foi obtida, dois conjuntos, (um que revela e outro que não revela o erro), foram derivados. Em tais casos o conjunto que revela o erro é apresentado na primeira linha da Tabela 3.3. Esta tabela também apresenta o número de casos de teste derivados para cada conjunto bem como a cobertura obtida utilizando-se as ferramentas POKE-TOOL e Proteum. A Tabela 3.4 apresenta os dados obtidos para os programas especiais, para os quais DT não foi aplicada.

A seguir apresentam-se uma síntese dos resultados obtidos e considerações que constituíram as principais motivações para combinar as técnicas estudadas com critérios estruturais de teste. O código fonte, descrição funcional, descrição dos erros, casos de teste, limitações encontradas e análise detalhada da revelação ou não do erro para cada técnica e cada programa podem ser encontrados em [VMJ96b].

Tabela 3.2: Resultados da Análise de Eficácia

Programa	Erro	Cond. Necessária. é Suficiente?	2. Pode revelar 3. Revela sempre			
			CBT	DT	BRO	BRE
max-1	comput.	não	2	2	2	2
max-2	dom.	sim	3	3	3	3
max-3	comput.	não	3	3	3	3
whi80-1	dom	sim	3	3	3	3
change-1	comput.	não	3	3	2	3
sum-1	dom.	sim	3	3	3	3
sum-2	comput.	não	3	3	3	3
triangle-1	dom.	não	2	-	2	2
voas-1	comput.	não	2	-	2	2

Tabela 3.3: Número de Casos de Teste Requeridos e Cobertura Obtida

Programa	Número de casos de teste/ Cobertura POKE e Proteum (%)											
	CBT			DT			BRO			BRE		
max-1	9	100	100	9	100	100	3	100	93	3	100	93
	9	100	100	9	100	98	3	100	89	3	100	90
max-2	8	100	100	9	100	98	3	100	88	3	100	94
max-3	7	100	100	9	100	100	3	100	85	3	100	88
whi80-1	15	100	100	24	100	99	5	95	96	5	95	94
change-1	8	100	100	9	100	99	3	100	90	3	100	96
							3	100	75			
sum-1	5	100	100	7	100	100	5	86	96	5	86	96
sum-2	5	100	100	6	100	100	5	100	99	5	100	99
TOTAL	57	100	100	73	100	99	30	97	95	30	97	96
	57	100	100	73	100	99	30	97	92	30	97	94

Tabela 3.4: Programas Especiais

Programa	Número de casos de teste / Cobertura POKE e Proteum (%)								
	CBT			BRO			BRE		
triangle-1	24	100	100	10	100	98	11	100	97
	21	100	100	10	100	98	11	100	97
voas-1	20	100	100	7	100	89	8	100	76
	19	100	100	7	100	88	8	100	75
TOTAL	44	100	100	17	100	98.5	19	100	86.5
	40	100	100	17	100	98	19	100	86

3.1.2 Principais Resultados/Motivações

As coberturas obtidas usando as ferramentas Proteum e POKE-TOOL para os conjuntos gerados respectivamente por CBT e DT foram sempre 100% pois essas técnicas foram aplicadas aos elementos requeridos por essas ferramentas. Note, no entanto que a média de cobertura da Proteum usando dados gerados com DT foi de 99%, bastante expressiva, assim como dados usando CBT sempre obtiveram cobertura de 100% na POKE-TOOL. Coberturas obtidas na Proteum e POKE-TOOL para os dados gerados por BRE e BOR também foram bastante altas. A técnica BRE apresenta uma vantagem com relação a BRO, uma cobertura mais alta, mas surpreendentemente isso não ocorre para os programas especiais.

BOR e BRE visam a encontrar erros de predicados, mas também foram eficazes em revelar os erros de computação para os programas *max-3*, *whi80-1* e *sum-2*. Mas para o programa *change-1* BRO não pôde garantir a revelação do erro. Nesse caso, BRE foi mais eficaz por tomar pontos a pequenas distâncias dos limites do domínio.

DT é mais poderosa que BRO e BRE mas é mais difícil de se automatizar devido as suas limitações. CBT também é muito poderosa porque ela pode descrever qualquer tipo de erro. No entanto, BOR e BRE requereram em, todos os casos, um número muito menor de casos de teste e se mostraram mais práticas com relação a automatização. Em [VMJ96b] são apresentadas tabelas que mostram a relação obtida nesse experimento, entre as técnicas BRO/BRE e mutantes para operadores relacionais e lógicos. Todos os mutantes, exceto os equivalentes, gerados pelos operadores: logical operator mutation, relational operator by logical operator, logical negation, logical operator by relational operator, relational operator mutation da Proteum, entre outros, foram mortos para todos os programas, utilizando-se os dados de teste gerados pelas técnicas BOR e BRE. As restrições BOR e BRE se aplicam a predicados compostos e são condições suficientes para a avaliação incorreta do predicado, incluindo mutação dos operadores relacionais e lógicos. Nesse sentido elas desempenham o mesmo papel que as restrições de predicados propostas por De Millo [DMO91] e descritas no capítulo anterior.

Nenhuma técnica pôde garantir a revelação do erro na presença de caminhos ausentes ou correção coincidente. Ainda, note na Tabela 3.4 que para esses programas, coberturas de 100% para ambas as ferramentas foram obtidas e mesmo assim não se garante a revelação do erro.

Um outro ponto interessante que pode ser visto na Tabela 3.2, diz respeito à relação entre o tipo de erro da versão incorreta e a dificuldade de se determinar relações necessárias e suficientes para a revelação do mesmo. Note que para a maioria das versões incorretas a condição de necessidade não foi suficiente. Observe que isso acontece para todos os erros de computação. Determinar condições suficientes para erros desse tipo é uma tarefa mais difícil. Na maioria das vezes a condição de suficiência está associada à condição de um caminho, ou seja, o erro pode ser revelado somente se um particular caminho no programa for executado. No entanto, a maior dificuldade encontrada

foi determinar a condição de suficiência para o programa *voas-1* que apresenta correção coincidente em muitos casos.

Idealmente a técnica DT deveria ser aplicada a todos os caminhos do programa, mas isso na maioria das vezes não é possível. O critério usado para escolher o conjunto de caminhos ao qual a técnica DT será aplicada, pode influenciar na eficácia da técnica; pode acontecer que nenhum caminho dentre os escolhidos seja capaz de revelar o erro. Nesse experimento, programas simples foram escolhidos, e a combinação DT com o critério todos-potenciais-usos (um critério baseado em fluxo de dados - data flow based criteria - DF) se mostrou bastante eficaz. Tudo indica que essa combinação é bastante útil para programas com laços, pois nesses casos, um número infinito de caminhos pode ser gerado e a técnica DF ajuda a escolher somente os "melhores" caminhos.

Isso pode ser exemplificado pelo programa da Figura 3.1. Suponha que esteja presente o erro de número 1: o comando incorreto $i = j * j$, no lugar de $i = i * j$. A Figura 3.2a apresenta os elementos requeridos e conjunto de caminhos selecionados pelos critérios estruturais CM caminhos linearmente independentes de MacCabe [McC76], CL todos-LCSAJ's [YM89], CR todos-ramos, CU todos-usos e CPU todos-potenciais-usos. Note que os pontos selecionados para os três primeiros critérios não revelaram o erro, pois o caminho 1 2 4 5 7 não foi selecionado. Os critérios todos-usos e todos-potenciais-usos revelaram o erro.

Isso constitui em uma motivação para aplicar a técnica DT juntamente com os critérios baseados em fluxo de dados. Isso porque eles selecionam os caminhos a serem executados não considerando apenas os aspectos de fluxo de controle do programa, mas como as variáveis são definidas e usadas.

Considere agora o programa da Figura 3.1 com o comando incorreto $i = i * i$ (erro 2). Os critérios anteriores continuam não garantindo a revelação do erro e o critério todos-usos deixa de requerer as associações $(2, 5, j)$ e $(3, 5, j)$ (Figura 3.2b). Portanto o caminho 1 2 4 5 7 pode não ser escolhido e o erro não ser revelado. O critério todos potenciais-usos, no entanto continua requerendo essas associações por considerar que um potencial uso de j poderia existir no nó 5, o caminho 1 2 4 5 7 é sempre selecionado e o erro sempre revelado.

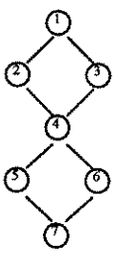
Os critérios todos potenciais-usos são mais exigentes que o critério todos-usos. Portanto escolher um critério mais exigente possibilita escolher um conjunto mais completo de caminhos para os quais será maior a probabilidade de revelar erros.

A aplicação da técnica Data Flow (DF) com a técnica Domain Testing também é vantajosa, pois posssibilita que pontos com maior probabilidade de revelar erros sejam escolhidos. O conjunto de dados de teste formado pelos pontos 2,7,C,G satisfaz o critério todos-potenciais-usos para o programa da Figura 3.1 e não revela o erro. A combinação Data Flow-DT é benéfica pois aumenta a capacidade de revelar erros de ambas as técnicas.

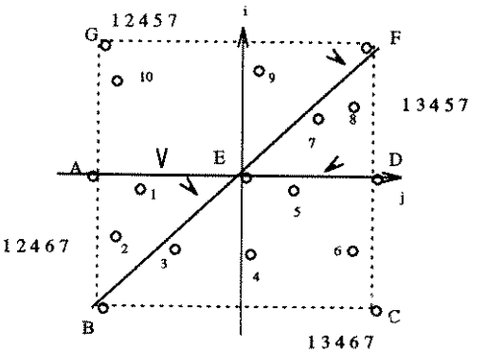
A aplicação de princípios de teste baseado em erros pode levar a um aumento da eficácia

```
void exemplo(i, j)
int i, j;
{
1     if (i>=j)
2       j=100;
3     else
3       j = i;
4     if (i>=0)
5       i= i*j;
6     else
6       i=i+1;
7     printf("%d",i);
}
```

Código Fonte



Grafo de Fluxo de Controle



Domínio do Programa

Ponto	(i,j)	Saída Esperada	Saída Obtida	Ponto	(i,j)	Saída Esperada	Saída Obtida
A	(0,-100)	0	0	3	(-50,-49)	-49	-49
B	(-100,-100)	-99	-99	4	(-99,0)	-98	-98
C	(-100,100)	-99	-99	5	(-1,50)	0	0
D	(0,100)	0	0	6	(-50,99)	-49	-49
E	(0,0)	0	0	7	(49,50)	49*49	49*49
F	(100,100)	100*100	100*100	8	(50,99)	50*50	50*50
G	(100,-100)	100*100	100*100	9	(99,0)	99*100	99*100
1	(1,-50)	0	0	10	(50,-99)	50*100	50*100
2	(-50,-99)	-49	-49				

Casos de Teste Requeridos pela Técnica DT

Figura 3.1: Exemplo - Motivação para Combinar Técnicas de Teste

Critério	Elementos Requeridos	Caminhos Selecionados	Casos de Teste
CM	conjunto linearmente independente de caminhos	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7	A B C D E F 1 2 3 4 5 6 7 8
CL	1 2 4, 1 3, 4 5 7, 4 6 7, 6 7 3 4 6, 3 4 5 7, 5 7	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7	A B C D E F 1 2 3 4 5 6 7 8
CR	1 2, 2 4, 1 3, 3 4, 4 5, 4 6, 5 7, 6 7	1 2 4 6 7, 1 3 4 5 7	A B D E F 1 2 3 5 7 8
CU	<1,(1,2),(i,j)>, <1,(1,3),(i,j)> <1,(4,5),i>, <1,(4,6),i> <1,3,i>, <1,6,i>, <1,5,j>, <2,5,j> <3,5,j>, <5,7,i>, <6,7,i>	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7, 1 2 4 5 7	A B C D E F G 1 2 3 4 5 6 7 8 9 10
CPU	<1,(1,2),(i,j)>, <1,(1,3),(i,j)> <1,(4,5),i>, <1,(4,6),i> <2,(4,5),j>, <2,(4,6),j> <3,(4,5),j>, <3,(4,6),j> <5,(5,7),i>, <6,(6,7),i>	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7, 1 2 4 5 7	A B C D E F G 1 2 3 4 5 6 7 8 9 10

a) Dados de Teste Requeridos para o Erro 1

Critério	Elementos Requeridos	Caminhos Selecionados	Casos de Teste
CM	conjunto linearmente independente de caminhos	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7	A B C D E F 1 2 3 4 5 6 7 8
CL	1 2 4, 1 3, 4 5 7, 4 6 7, 6 7 3 4 6, 3 4 5 7, 5 7	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7	A B C D E F 1 2 3 4 5 6 7 8
CR	1 2, 2 4, 1 3, 3 4, 4 5, 4 6, 5 7, 6 7	1 2 4 6 7, 1 3 4 5 7	A B D E F 1 2 3 5 7 8
CU	<1,(1,2),(i,j)>, <1,(1,3),(i,j)> <1,(4,5),i>, <1,(4,6),i> <1,3,i>, <1,5,i>, <1,6,i> <5,7,i> <6,7,i>	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7	A B D E F 1 2 3 5 7 8
CPU	<1,(1,2),(i,j)>, <1,(1,3),(i,j)> <1,(4,5),i>, <1,(4,6),i> <2,(4,5),j>, <2,(4,6),j> <3,(4,5),j>, <3,(4,6),j> <5,(5,7),i>, <6,(6,7),i>	1 2 4 6 7, 1 3 4 5 7 1 3 4 6 7, 1 2 4 5 7	A B C D E F G 1 2 3 4 5 6 7 8 9 10

b) Dados de Teste Requeridos para o Erro 2

Figura 3.2: Dados de Teste Requeridos para as Duas Versões Incorretas

dessas técnicas. Os pontos selecionados aplicando-se a combinação DF-DT, também poderiam satisfazer certas restrições que descrevem erros típicos. Portanto uma nova combinação DF-DT-CBT. Essa nova combinação daria mais poder ao DT pois, os pontos escolhidos também teriam maior probabilidade de revelar erros de computação.

O programa *max* (Figura 3.3) contém um erro de computação no nó 1. O comando correto é $max = m$. A aplicação da técnica DF-DT não garante a revelação do erro. Isso dependerá da escolha entre os pontos OFF 1, 1' e 1". A condição de necessidade associada ao erro é $(max = abs(m)) \neq (max = m)$. Se para escolher entre esses pontos essa condição for considerada o erro será revelado.

A condição de suficiência para revelar esse erro é $n < m$, que é a condição do caminho 1 3 4. Determinar condições de suficiência é uma questão indecidível. Muitos autores [DMO91] consideram que uma vez satisfeita a condição de necessidade, a condição de suficiência será também satisfeita. No entanto, como mostrado acima, isso não ocorreu no estudo realizado. A combinação DF-DT-CBT pode contribuir para que os casos nos quais a condição de suficiência é satisfeita, ocorram mais freqüentemente. Mais de um dado de teste será gerado satisfazendo a condição de necessidade. Além disso, eles satisfarão diferentes condições, as condições dos diferentes caminhos requeridos, que poderão resultar em condições suficientes. Para o exemplo *max*, pontos que satisfazem $m >= n, m < n$, juntamente com $abs(m) \neq m$ serão escolhidos.

3.2 Critérios de Teste Estrutural Baseados em Restrições

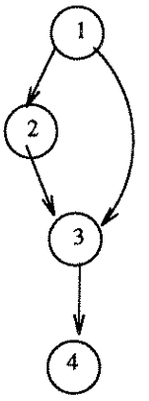
Critérios de teste estrutural derivam elementos do programa fonte em teste a serem exercitados. Uma maneira de aumentar a capacidade desses critérios em revelar erros e combinar a técnica estrutural com os principais fundamentos das técnicas de geração de dados sensíveis a erros, é associar a cada elemento requerido 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 restrições que descrevem erros e portanto têm alta probabilidade de revelar um erro ainda não descoberto.

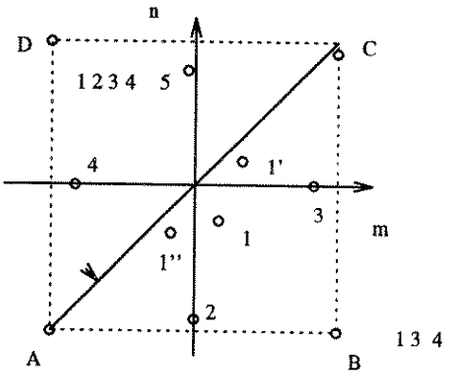
A seguir, são definidos os Critérios Restritos (ou Baseados em Restrições) para os dois grupos de critérios estruturais mais conhecidos: critérios baseados em fluxo de controle e critérios baseado em fluxo de dados; mas a idéia pode ser aplicada a qualquer critério de teste estrutural. Critérios restritos (ou baseados em restrições) requerem elementos restritos. Diz-se que o elemento requerido é um elemento restrito porque seu domínio de entrada foi restringido. Ao se associar uma restrição a um elemento, o domínio de entrada do elemento restrito gerado fica restrito aos pontos que satisfazem a restrição dada.

```
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  }
```

Código Fonte



Grafo de Fluxo de Controle



Domínio do Programa

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

Elementos Requeridos pelo Critério Todos-Potenciais-Usos

Figura 3.3: Programa para Calcular o Máximo entre Dois Números

1. Critérios Restritos Baseados em Fluxo de Controle

   - **Critério todos-nós restritos**: associa-se a cada nó 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**: associa-se a cada ramo 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**: associa-se a cada caminho completo uma ou mais restrições. Um caminho restrito é dado pelo par (P,C), e será coberto se C for satisfeita durante a sua execução.
   - **Critério todos-LCSAJ's restritos**: associa-se a cada LCSAJ uma ou mais restrições. Um LCSAJ restrito é dado pelo par (L,C) e será coberto se um caminho que cobre o LCSAJ for executado, e se C for satisfeita durante a sua execução.

2. Critérios Restritos Baseados em Fluxo de Dados

   - **Critério todos-usos restritos**: associa-se a cada associação requerida 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**: associa-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**: associa-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**: associa-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 uma 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**: associa-se a cada potencial-du-caminho 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.

Também definem-se os Critérios Restritos Executáveis (ou *Criterios**), requerendo-se apenas os pares (E,C), (elemento, restrição), executáveis para cada critério.

• Sobre as Restrições

As restrições a serem associadas aos elementos requeridos pelos critérios estruturais definidos na seção anterior, podem descrever qualquer tipo de erro. Para o estabelecimento dessas restrições, diferentes princípios de diferentes técnicas de teste podem ser considerados. Nessa seção são apresentadas algumas possíveis restrições e uma discussão de como bem utilizá-las.

A todo elemento requerido está associado uma restrição implícita, chamada restrição de cobertura, que é dada pela condição de um caminho completo que cobre o elemento. As restrições associadas ao elemento serão restrições de necessidade, ou seja, descreverão condições necessárias para revelar o erro que descrevem. Uma mesma restrição poderá estar associada a mais de um elemento requerido, poderá ser satisfeita conjuntamente com mais de uma restrição de cobertura, portanto maior será a probabilidade dessa conjunção satisfazer a condição de suficiência, e o erro ser revelado.

Possíveis restrições foram derivadas, considerando as seguintes técnicas de teste: análise de mutantes, BOR/BRO/BRE, Análise do Valor Limite e Teste de Computações. Uma síntese das restrições de necessidade, derivadas para cada técnica, é apresentada na Tabela 3.5. Essa tabela apresenta somente as restrições que foram selecionadas a partir de um estudo de possíveis restrições derivadas utilizando os princípios de cada técnica.

Algumas das restrições aqui apresentadas, fazem parte das restrições geradas para os operadores de mutação da ferramenta Proteum, utilizados no experimento da Seção 3.1 e disponíveis no Apêndice A. Elas são aproximações da condição de suficiência, e têm o objetivo de produzir um resultado final incorreto na avaliação de um predicado ou expressão, se um erro estiver presente. "Statement Mutations", na maioria dos casos, não possuem restrições de necessidade associadas. Basta satisfazer a condição de alcançabilidade do comando mudado. Os mutantes derivados dessas mutações, são geralmente mortos por dados de teste que satisfazem restrições associadas a outros tipos de mutação, e por dados de teste gerados para satifazer critérios baseados em fluxo de dados.

A Tabela 3.5 foi elaborada segundo os critérios já mencionados. Mas além desses, é muito importante considerar outros critérios tais como: número de elementos não executáveis; tipos de erros aos quais o programa está sujeito; facilidade de implementação, que pode ser dada pela quantidade de informação necessária sobre o programa.

As restrições de cobertura de cada elemento requerido não serão determinadas na hora de gerar elementos restritos, ou seja na hora de se estabelecer o par (E,C). No entanto, elas influenciarão a geração do dado de teste e poderão tornar os elementos restritos não executáveis. Isto é, a satisfação de C poderá implicar a não satisfação da restrição de cobertura. Reduzir o número de

Tabela 3.5: Restrições Possíveis

Técnica Relacionada	**Descrição** Operadores-Proteum	**Restrição de Necessidade**
Mutação em Comandos	do-while (CT) replacement by while	CT = false
	trap on if condition (CT)	CT = false, CT = true
	while (CT) replacement by do-while	CT = false
Mutação em Operadores	operator by other operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$, $(e_1 op_1) \neq (e_1 op_2)$ $(e_1 op_1) \neq (op_1 e_1)$, $(op_1 e_2) \neq (op_2 e_2)$ $(op_1 e_2) \neq (e_2 op_1)$, $e_1 \neq op e_2$
Mutação em Variáveis	domain traps for scalars	$X = 0, X < 0, X > 0$ $A[e_1] > 0, A[e_1] < 0, A[e_1] = 0$ $S.c = 0, S.c > 0, S.c < 0$
	mutate array reference	$A[e_1] \neq B[e_1]$
	mutate pointer reference	$p \neq q$
	mutate scalar reference	$A[e_1] \neq X$, $S.c \neq X$, $(*p) \neq X$, $Y \neq X$
	mutate structure reference	$S \neq T$, $S.c \neq T.c$
	mutate component structure replac.	$S.c_1 \neq S.c_2$
	twiddle mutation	PRED(x) $\neq$ x, SUC(x) $\neq$ x
Mutação em Constantes	constant for constant replacement	$C_1 \neq C_2$
	constant for scalar replacement	$X \neq C$
	required constant replacement	$X \neq 0, X \neq -1, X \neq 1$
BOR/BRO	$C_1 or C_2$	$F(C) = S_{1f} \% S_{2f}$ $T(C) = \{S_{2t} * \{f_2\} \$ \{\{f_1\} * S_{2t}\}$ onde $f_1 \in S_{1f} e f_2 \in S_{2f} e (f_1, f_2) \in F(C)$
	$C_1 and C_2$	$T(C) = S_{1t} \% S_{2t}$ $F(C) = \{S_{1f} * \{t_2\} \$ \{\{t_1\} * S_{2f}\}$ onde $t_1 \in S_{1t} e t_2 \in S_{2t} e (t_1, t_2) \in T(C)$
AVL	A[dim]	$p = null$ $e_1 = dim, e_1 = dim + 1, e_1 = dim - 1$ X = +maxint, X = -maxint X = +maxint+1, X = -maxint-1 X = +maxint-1, X = -maxint+1 X = +maxfloat, X = -maxfloat X = +maxfloat+1, X= -maxfloat+1 X = +maxfloat-1, X = -maxfloat+1 X = endfile, X = endline, X = '\0'
Teste de Computações		$e_1 = 0, e_1 > 0, e_2 < 0, e_1 =$ null e_1=+maxint, e_1= -maxint e_1=+maxint+1, e_1= -maxint-1 e_1=+maxint-1, e_1= -maxint+1 e_1=+maxfloat, e_1= -maxfloat e_1=+maxfloat+1, e_1= -maxfloat-1 e_1=+maxfloat-1, e_1= -maxfloat+1 e_1=endfile, e_1=endline, e_1='\0'

elementos restritos não executáveis requeridos pode implicar em uma análise de caminhos candidatos a cobrir o elemento, ou seja determinar a restrição de cobertura. Isso requer grande quantidade de conhecimento sobre o programa e implica um custo adicional. Considerando os aspectos de custo e executabilidade, sugere-se estruturar a escolha de restrições em diferentes níveis. A cada nível, novas restrições vão sendo acrescentadas e uma maior quantidade de informação sobre o programa sendo necessária.

Tendo como base as restrições apresentadas na Tabela 3.5, e considerando fatores custo e executabilidade de elementos, são detalhados três diferentes níveis. Dada a informação disponível em cada nível, as possíveis restrições, para a aplicação do critério todos-potenciais-usos são apresentadas. Esse critério foi escolhido por favorecer a ilustração, pois a quantidade de restrições aplicadas em cada nível é maior para os elementos por ele requeridos.

Nível 1: informação disponível: a associação na forma $(i,(j,k),x)$ ou (i,j,x). Domain traps que são operadores de mutação da ferramenta Proetum, e A.V.L. (Análise do Valor Limite) são aplicadas 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 tal maneira que a intersecção do conjunto de caminhos candidatos a cobrir as associações seja diferente do conjunto vazio. Restrições de necessidade associadas ao operador de mutação Variable Mutation são aplicadas às variáveis $x_1...x_n$. Pode-se também ter outras restrições além de $X \neq Y$, tais como $X < Y, X > Y, X = Y, X < Y + 1, X + 1 > Y$, etc.

Nível 3: informação disponível: o predicado no nó j, para uma associação do tipo (i,(j,k),x). No caso do critério todos p-usos a existência do predicado é garantida. Se o predicado não existir, nenhuma restrição será gerada. 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 tal maneira que o arco (j,k) seja executado.

Em [VMJ96c] são descritos outros níveis para a escolha das restrições, onde a informação disponível pode ser um conjunto de caminhos candidatos a cobrir uma associação requerida, ou um caminho completo. Nesse caso, técnicas BOR/BRO/BRE são aplicadas aos predicados encontrados ao longo dos caminhos, escolhendo-se conjuntos T e F que garantam que eles sejam executados. A escolha de uma técnica em um nível não implica que as outras técnicas deste nível devam ser selecionadas. A estruturação em níveis, apesar de hierárquica, não implica que para a seleção de um nível, os níveis anteriores devam ter sido utilizados. A estruturação tem o objetivo de facilitar a escolha das restrições, especificando a quantidade de informação necessária sobre o programa em cada nível. Ela diz que a utilização de uma técnica em um nível maior implica num maior custo em termos de processamento para a geração das restrições.

Tabela 3.6: Restrições Possíveis em Cada Nível

Inform. Dispon.	**Técnica**	**Descrição**	**Condição de Necessidade**
(i,j,k,x) **Nível 1**	Mutação Comandos	domain traps	$X = 0, X < 0, X > 0$ $A[e_1] > 0, A[e_1] < 0, A[e_1] = 0$ $S.c = 0, S.c > 0, S.c < 0$ $p = null$
um conj.: (i,j,x_1) (i,j,x_2) .... (i,j,x_n) **Nível 2**	Mutação Variáveis	mut. array ref mut. pointer ref mut. scalar ref mut. str ref mut. c. str repl	$A[e_1] \neq B[e_1]$ $p \neq q$ $A[e_1] \neq X$ $S.c \neq X$ $(*p) \neq X$, $Y \neq X$ $S \neq T$, $S.c \neq T.c$ $S.c_1 \neq S.c_2$
(i,(j,k),x) pred. em (j,k) **Nível 3**	BRO/BRE	$C_1 or C_2$	$F(C) = S_{1f} \% S_{2f}$ $T(C) = \{S_{2t} * \{f_2\} \$ \{\{f_1\} * S_{2t}\}$ onde $f_1 \in S_{1f}$ e $f_2 \in S_{2f}$ e $(f_1, f_2) \in F(C)$
		$C_1 and C_2$	$T(C) = S_{1t} \% S_{2t}$ $F(C) = \{S_{1f} * \{t_2\} \$ \{\{t_1\} * S_{2f}\}$ onde $t_1 \in S_{1t}$ e $t_2 \in S_{2t}$ e $(t_1, t_2) \in T(C)$

3.3 Aplicação dos Critérios Baseados Em Restrições - Projeto Piloto

Os programas do experimento descrito na Seção 3.1, exceto os programas especiais (*triangle* e *voas*), foram utilizados para a aplicação dos critérios baseados em restrições. Foram considerados três níveis de restrições, especificados na Tabela 3.6. Por exemplo, os elementos restritos gerados considerando esses níveis, para o programa da Figura 3.3, são apresentados na Tabela 3.7.

Primeiramente foram derivados, a partir da versão correta, os elementos requeridos pelo critério todos-potenciais-usos (PU) e todos potenciais-usos-restritos (PU-R). Dados de teste adequados ao critério todos-potenciais-usos foram gerados e foi verificada a eficácia destes pela execução das versões incorretas. O mesmo foi realizado para o critério todos potenciais-usos-restritos, mas por testadores independentes. Além disso, os casos de teste para ambos critérios foram submetidos à ferramenta Proteum. Determinados os mutantes equivalentes, foi obtida a cobertura, tendo-se anotado também o número de casos de teste que realmente contribuiram para a morte de mutantes.

A Tabela 3.8 mostra a cobertura obtida para ambos os critérios, bem como o número de casos de teste necessários. O critério PU-R requereu 6.6 vezes mais elementos que o critério PU,

Tabela 3.7: Associações Restritas para o Programa *max*

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

no entanto, devido ao número de elementos não executáveis ser maior, o número de casos de teste foi apenas 2.7 vezes maior. Isso mostra que o número de casos de teste necessários não tende a crescer tanto quanto o número de elementos requeridos. Também, em comparação com a Tabela 3.3, o número de casos de teste necessários foi menor que os exigidos pelas técnicas CBT (37) e DT (59) (primeira entrada da tabela, para os conjuntos que revelam o erro).

Todos os erros foram revelados pelo critério PU-R (Tabela 3.9), até mesmo o erro de *max* é garantidamente revelado pelo PU-R para o conjunto de restrições estabelecido, o que não acontece para nenhuma técnica aplicada na Seção 3.1 e para o critério PU. No total o critério PU-R revelou 28.57% a mais de erros que o critério PU. Considerando-se apenas erros de computação, o critério PU revelou apenas 50% dos erros.

Outro ponto interessante é que a cobertura obtida na Proteum com os dados gerados para o PU-R, foi sempre maior do que a obtida com os dados gerados para o PU (Figura 3.10), numa média de 99.25%, sendo 100% para os programas *max* e *whi80*, com um número menor ou igual de casos de teste gerados pela técnica CBT (Ver Tabela 3.3). Esses resultados mostram o aumento da eficácia dos critérios baseados em fluxo de dados, através da utilização de restrições e de uma maneira mais barata, em termos do número de casos de teste, do que o critério análise de mutantes.

3.4 Análise de Propriedades dos Critérios Baseados em Restrições

3.4.1 Complexidade dos Critérios

A complexidade de um critério C é definida como o número de casos de teste requerido pelo critério no pior caso, ou seja, dado um programa qualquer P, se existir um conjunto de casos de teste T que seja C-adequado para P, a cardinalidade de T é menor ou igual à complexidade do critério C.

É difícil estabelecer a complexidade de um critério restrito, visto que esta é completamente dependente do conjunto de restrições escolhidas e/ou nível considerado. Mas com certeza ela será sempre maior ou igual à complexidade do critério estrutural correspondente. Para se obter a

Tabela 3.8: Resultados da Análise de Cobertura dos Critérios PU e PU-R

Programa	Critério	Elem. Req	Elem. Exec.	Elem. NExec.	Cobertura POKE-TOOL	No.Casos Teste
change	PU	4	4	0	100	2
	PU-R	27	23	4	85.19	8
max	PU	3	3	0	100	2
	PU-R	23	23	0	100	7
sum	PU	8	8	0	100	3
	PU-R	53	30	23	56.60	5
whi80	PU	20	19	1	95.00	6
	PU-R	130	108	22	88.67	15
Total	PU	35	34	1	97.14	13
	PU-R	233	184	49	78.97	35

Tabela 3.9: Eficácia Obtida para os Critérios PU e PU-R

Programa	Critério	Erro Comp.	Erro Dom.	Total
change	PU			1
	PU-R	change-1		1
max	PU	max-3	max-2	2
	PU-R	max-1, max-3	max-2	3
sum	PU	sum-2	sum-1	2
	PU-R	sum-2	sum-1	2
whi80	PU		whi80-1	1
	PU-R		whi80-1	1
Total	PU	2(4) 50%	3 100%	5(7) (71.43%)
	PU-R	4 100%	3 100%	7 100%

Tabela 3.10: Cobertura dos Conjuntos PU e PU-R Adequados na Ferramenta Proteum

Programa	Critério	Cobertura	No.CasosTeste
change	PU	90	2
	PU-R	98	7
max	PU	85	2
	PU-R	100	7
sum	PU	97	3
	PU-R	99	5
whi80	PU	97	6
	PU-R	100	15
Total	PU	92.25	13
	PU-R	99.25	34

complexidade de um critério restrito, primeiramente, é necessário fazer uma análise do pior caso com relação às técnicas utilizadas em cada nível, para derivar as restrições. Abaixo, são feitas algumas considerações com relação às técnicas utilizadas para derivar as restrições considerando-se os três níveis descritos na Tabela 3.6.

Nível 1: no pior caso, são necessários 3 casos de teste para a associação $(i,(j,k),var)$ correspondente, fazendo $var > 0$, $var = 0$ e $var < 0$.

Nível 2: no pior caso são necessários $k(k+1)/2$ casos de teste para a associação $(i,(j,k),$ $\{v_1, v_2, .., v_k\})$, onde k é o número de variáveis envolvidas.

Nível 3: no pior caso, aplicando-se as técnicas BOR e BRO, serão derivadas de acordo com Tai [Tai93], para um predicado p com n operadores booleanos (do tipo AND ou OR):

BOR: $n + 2$ restrições

BRO: $2 * n + 3$ restrições.

A partir do nível e técnica aplicados e do número de elementos requeridos pelo correspondente critério estrutural, calcula-se a complexidade do critério restrito. Por exemplo, a complexidade do critério todos ramos-restritos, aplicando-se a técnica BRO no Nível 3 é:

$\sum_{i=1}^{t}(2 * n_i + 3)$, onde t é o número de comandos de decisão do programa e n_i é o número de operadores AND/OR do predicado associado ao comando i, se o predicado existir.

Os critérios todos-usos restritos e todos-potenciais-usos restritos possuem complexidade exponencial, visto também ser exponencial a complexidade dos correspondentes critérios estruturais. No entanto, tem-se observado em estudos empíricos [Wey90], [Mal91] uma complexidade linear para os critérios baseados em fluxo de dados, em função do número de comandos de decisão do programa. É necessário observar a complexidade empírica dos critérios-restritos.

3.4.2 Análise de Inclusão

Considere que para todo elemento requerido E é derivado um conjunto de restrições C, tal que $(true) \in C$. Dessa maneira, ao se cobrir o elemento restrito requerido (E,true), o elemento E é coberto, garantindo assim que um critério restrito inclua o critério correspondente que o originou. A relação de inclusão entre os Critérios Estruturais Restritos e os correspondentes Critérios Estruturais é dada pela Figura 3.4a.

A relação entre os Critérios Restritos é dependente do conjunto de Restrições escolhido. Se o conjunto nao for o mesmo, os critérios são incomparáveis. No entanto, se um conjunto comum de restrições é associado a cada elemento requerido, para todos os critérios estruturais, a fim de

derivar os critérios restritos correspondentes, tem-se a relação dada na Figura 3.4b. Essa relação é discutida mais detalhadamente em [VMJ96c].

Assim como foi ressaltado no Capítulo 2 e em [HL90], [VMJ95b], se não for considerada a suposição feita por Frankl e Weyuker [FW88] de que cada predicado e cada comando deve fazer referência a pelo menos uma variável, o critério todos-usos deixa de incluir o critério todos-ramos. Nesse caso, todos-usos-restritos também não inclui todos-ramos restritos (Figura 3.4c). Nada é alterado para os Critérios Potenciais Usos e correspondentes critérios restritos.

Caminhos não executáveis alteram a relação de inclusão entre os critérios. Os critérios todos-du-caminhos e todos-usos deixam de incluir o critério todos-ramos [Mal91]. Se um elemento E é não executável todos os elementos restritos da forma (E,R) serão não executáveis. A ordem entre os Critérios Estruturais e seus correspondentes Critérios Restritos não é afetada, na presença de elementos não executáveis, independentemente do conjunto de restrições. Isso pode ser visto na Figura 3.5a. Veja na Figura 3.5 a relação entre os Critérios Restritos Executáveis (Figura 3.5a considerando um conjunto diferente de restrições e Figura 3.5b considerando o mesmo conjunto de restrições). Note que os critérios (todos-du-caminhos restritos)* e (todos-usos-restritos)* não incluem o critério (todos-ramos-restritos)*.

No Apêndice B encontram-se as provas das relações apresentadas. Maiores detalhes e suposições feitas durante a elaboração dessas provas podem ser encontradas em [VMJ96c, Mal91]

3.4.3 A Relação de Inclusão e a Eficácia dos Critérios

A relação de inclusão entre critérios diz muito pouco sobre a capacidade destes em revelar erros. A hierarquia não é preservada quando o fator eficácia é considerado. O fato do critério C_1 incluir o critério C_2, não garante que T_1 (um conjunto de teste que satisfaz C_1) revelará todos os erros revelados por T_2 (um conjunto de teste que satisfaz C_2). Considere o programa *max-2* da Figura 3.6. A Tabela 3.11 apresenta os elementos requeridos para os critérios todos-ramos e todos-potenciais-usos. Note nas Tabelas 3.12 e 3.13 que entre os dados de teste gerados somente o caso de teste 13 selecionado para o critério todos-ramos revela o erro; portanto, a hieraquia não é preservada; somente o critério mais fraco revelou erro. Isso porque a eficácia de T_1 e de T_2 dependerá obviamente da estratégia utilizada para gerá-los. Para preservar a hierarquia entre os critérios, ou seja, dizer que C_1 é melhor que C_2 segundo o fator eficácia, é necessário que T_1 ($C_1 - adequado$) revele pelo menos todos os erros revelados por T_2 ($C_2 - adequado$).

Considere o conjunto-1 de elementos restritos derivados para os critérios todos-ramos e todos-potenciais-usos utilizando o conjunto de restrições-1 da Tabela 3.11. A restrição de necessidade ($m < 0$) estará associada a vários elementos e necessariamente a restrição de suficiência ($m > n$) será satisfeita pela condição do arco (1,3). O caso de teste 13 satisfaz essa restrição e o erro será garantidamente revelado por todos-ramos-restritos-1 e todos-potenciais-usos-restritos-1

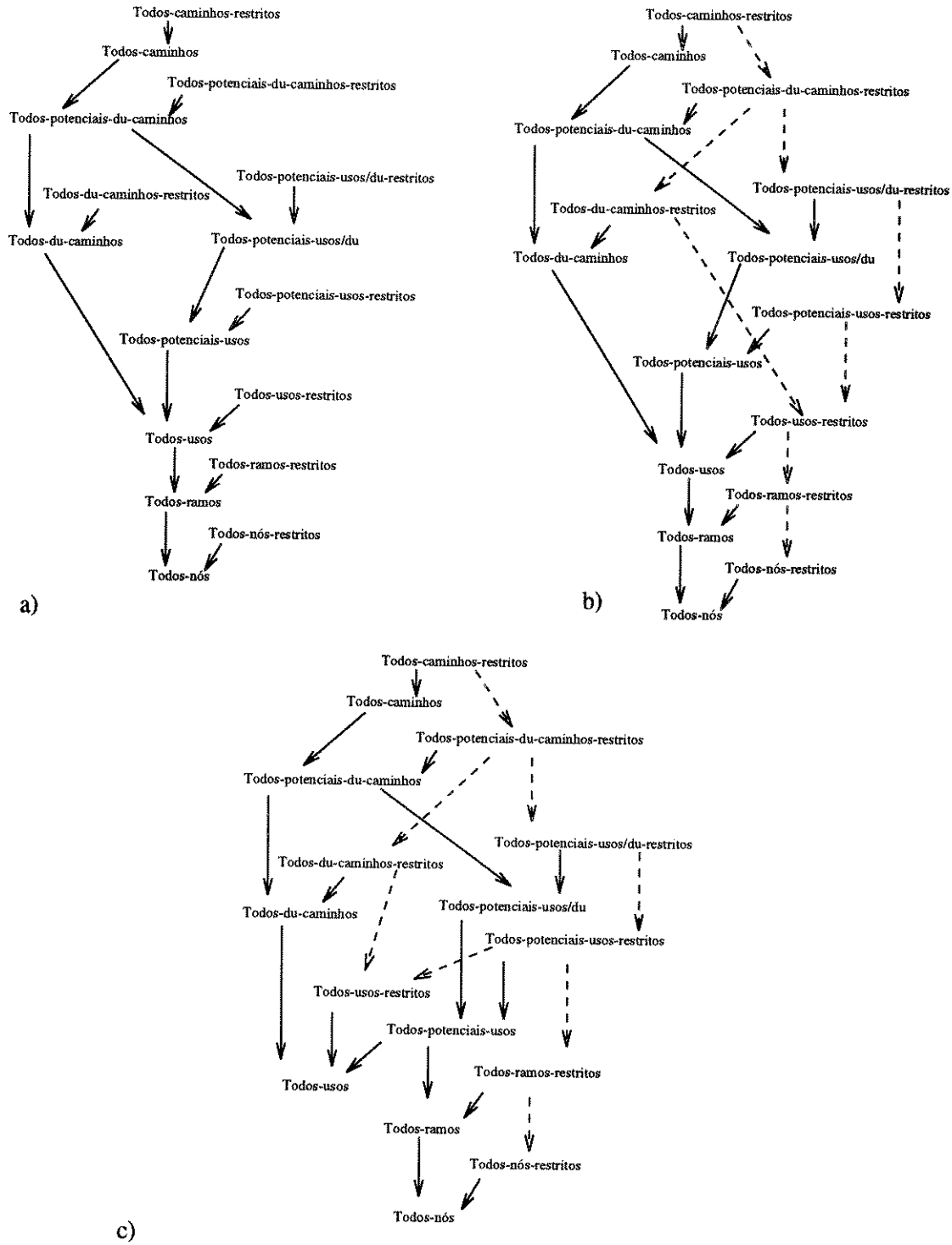


Figura 3.4: Relação de Inclusão entre os Critérios Restritos e os Correspondentes Critérios Estruturais: a) Considerando diferentes conjuntos de restrições; b) Considerando o mesmo conjunto de restrições; c) Consideranto um conjunto comum de restrições e a linguagem C

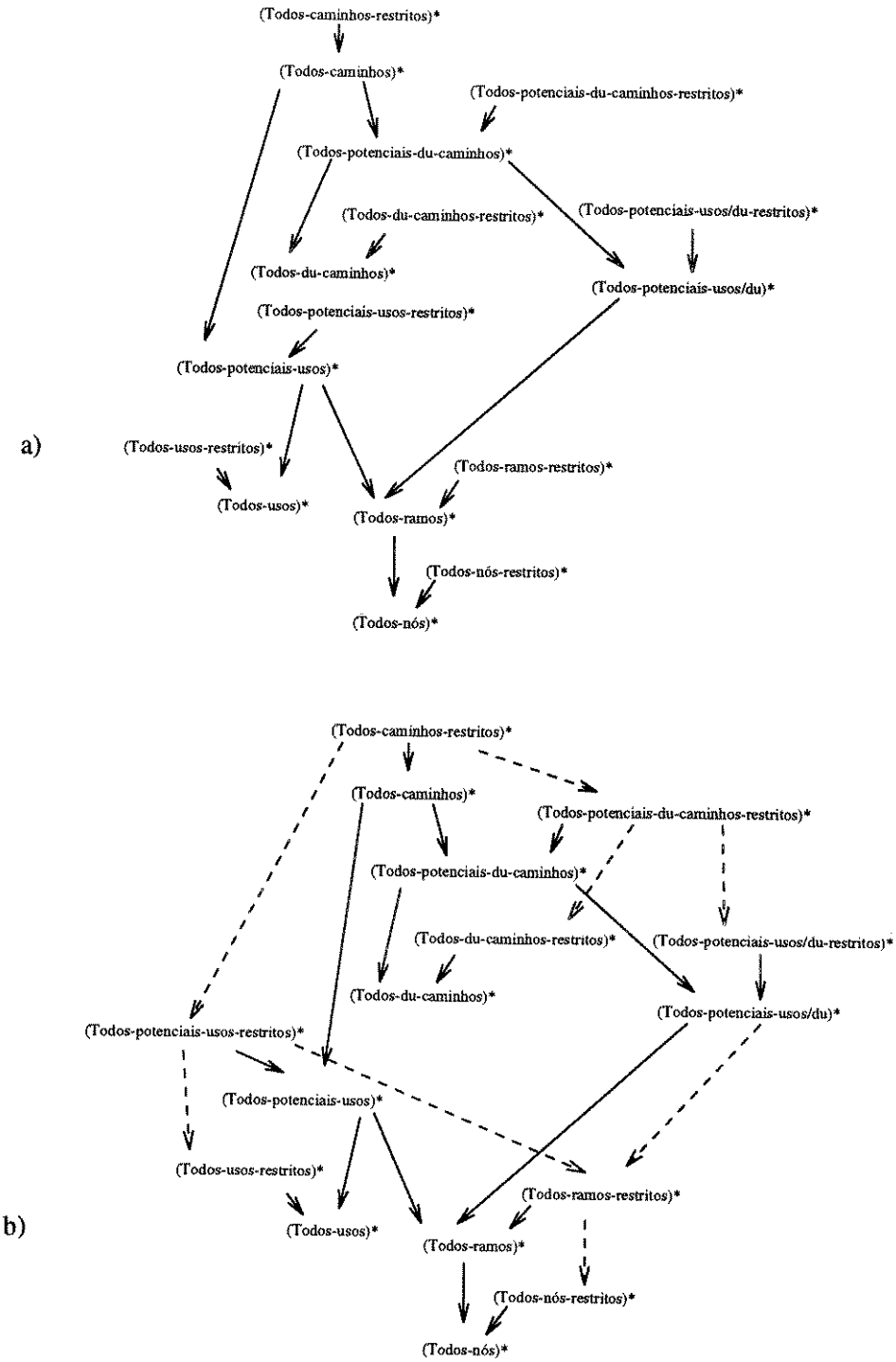


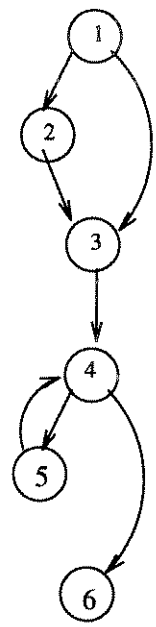
Figura 3.5: Relação de Inclusão entre os Critérios Restritos e os Correspondentes Critérios Estruturais na Presença de Caminhos Não Executáveis: a) Considerando diferentes conjuntos de restrições; b) Considerando o mesmo conjunto de restrições

Tabela 3.11: Elementos Requeridos pelos Critérios Restritos para o Programa *max-2*

todos-ramos	todos-pot-usos	restrições 1	restrições 2
(1,2)	(1,(1,3),{ max, m, n })	true, (m>0), (m<0),(m==0)	true, (n>0), (n<0),(n==0)
(1,3)	(1,(4,6),{ max, m, n })	(n>0), (n<0),(n==0)	
(4,5)	(1,(4,6),{ m, n })		
(4,6)	(1,(5,4),{ max, m, n })		
	(1,(5,4),{ m, n })		
	(1,(4,5),{ max, m, n })		
	(1,(4,5),{ m, n })		
	(1,(1,2),{ max, m, n })		
	(2,(4,6),{ max })		
	(2,(5,4),{ max })		
	(2,(4,5),{ max })		
	(3,(4,6),{ i })		
	(3,(4,5),{ i })		
	(5,(4,6),{ i })		
	(5,(4,5),{ i })		

```
   void maximo()
   {
    int i,max,m,n;
1    scanf{"%f", &m);
1    scanf("%f ", &n);
1    max = abs(m);
1    if (n>=m)
2      max = n;
3    printf("%f", max);
3    i =1
4    while (i<abs(max)
5    {
5      printf("*");
5      i++;
5    }
6 }
```

Código Fonte



Grafo de Fluxo de Controle

Figura 3.6: Programa *max-2*

Tabela 3.12: Casos de Teste para o Programa *max-2*

número	dado de teste valor de (m,n)	saída esperada	saída obtida	caminho percorrido
1	(0,4)	4 ***	4 ***	1 2 3 4 5 4 5 4 5 4 6
2	(-2,4)	4 ***	4 ***	1 2 3 4 5 4 5 4 5 4 6
3	(5,6)	6 *****	6 *****	1 2 3 4 5 4 5 4 5 4 5 4 6
4	(0,0)	0	0	1 2 3 4 5
5	(0,-1)	0	0	1 3 4 6
6	(5,4)	5 ****	5 ****	1 3 4 5 4 5 4 5 4 5 4 6
7	(5,0)	5 ****	5 ****	1 3 4 5 4 5 4 5 4 5 4 6
8	(0,-2)	0	0	1 3 4 6
9	(-3,-2)	-2	-2	1 2 3 4 5 4 6
10	(-1,0)	0	0	1 2 3 4 6
11	(-1,1)	1	1	1 2 3 4 6
12	(5,-2)	5 ****	5 ****	1 3 4 5 4 5 4 5 4 5 4 6
13	(-3,-4)	-3 **	3 **	1 3 4 5 4 5 4 6

(Veja os dados da Tabela 3.13). A hierarquia é preservada nesse caso.

Existem casos nos quais a preservação da hierarquia não é garantida. Pode ser que o conjunto de restrições escolhido não inclua uma condição de necessidade capaz de revelar o erro, como por exemplo, se forem associadas aos critérios o conjunto de restrições-2 (Tabela 3.11). Note que os critérios restritos podem ser satisfeitos, e o erro ser ou não revelado, e a hierarquia ser ou não preservada, utilizando os conjuntos derivados para satisfazer todos-ramos-restritos-2 e todos-potenciais-usos-restritos-2 da Tabela 3.13. Outro fator que influirá é a restrição de suficiência. Ela pode não estar associada à nenhuma condição de caminho; podendo existir até mesmo casos de correção coincidente, mas tudo indica que essas situações acontecem raramente.

Percebeu-se durante o experimento descrito na Seção 2, que quanto mais exigente o critério um maior número de condições de caminhos serão satisfeitas conjuntamente com as restrições associadas aos elementos restritos e maior a probabilidade de também satisfazer a restrição de suficiência. Portanto, a utilização de critérios restritos, associando-se aos elementos requeridos um conjunto comum de restrições, contribui em muitos casos para a preservação da hierarquia com relação à revelação de erros descritos por essas restrições.

3.5 Considerações Finais

Nesse capítulo foram introduzidos os Critérios Restritos, ou *Critérios Baseados em Restrições*. Foi descrito um experimento de motivação para a definição desses critérios; foram discutidas

Tabela 3.13: Casos de Teste Selecionados para Cada Critério

critério	número do caso de teste
todos-ramos	5 13
todos-potenciais-usos	1 4 5 6
todos-ramos-restritos-1	1 2 3 4 5 6 7 9 13
todos-potenciais-usos-restritos-1	1 2 3 4 5 6 7 9 10 11 12 13
todos-ramos-restritos-2	1 2 3 4 6 7 9 13
todos-potenciais-usos-restritos-2	1 4 5 6 7 9 11 12
todos-ramos-restritos-2	1 4 6 7 8 11
todos-potenciais-usos-restritos-2	1 4 6 7 8 9 11 12 13

suas propriedades, e apresentados resultados de um projeto piloto, aplicando esses critérios.

Os critérios baseados em restrições combinam princípios de diferentes estratégias de teste com critérios estruturais. A idéia é associar a cada elemento requerido por um determinado critério de teste estrutural, uma restrição de necessidade que descreve um erro típico em programação, criando assim o elemento restrito correspondente. O elemento restrito é coberto pelo caso de teste que executa o caminho que cobre o elemento requerido, e cuja execução, satisfaz a restrição. Essa idéia é devida a um experimento com diferentes estratégias de geração de dados de teste e está fortemente motivada pelo fato de que muitas vezes a combinação da restrição de necessidade com a condição do caminho resulta em uma condição de suficiência para que o erro seja revelado.

O experimento de motivação evidenciou alguns outros resultados relativos a aplicação das estratégias utilizadas e que devem ser considerados para a escolha das restrições:

- alta eficiência em termos de revelação de erros e de cobertura das ferramentas POKE-TOOL e Proteum, respectivamente pelas estratégias CBT e DT
- praticidade das técnicas BOR/BRO; elas requerem um número menor de casos de teste e incluem mutação do operador lógico e do operador relacional; são restrições suficientes para a avaliação incorreta do predicado composto, incluem restrições de predicado da estratégia CBT.
- dificuldades de aplicação do DT em domínios discretos e não lineares.
- limitações das técnicas, pela não garantia de revelação dos erros na presença de correção coincidente e caminhos ausentes

As restrições escolhidas para gerar os elementos restritos, poderão descrever outros tipos de erros comuns em diferentes situações, por exemplo, associar restrições especiais às variáveis que são passadas como parâmetros para detectar erros de interface. Essas restrições podem ser derivadas de operadores de mutação para teste de integração definidos em [Dea96] e [Del97]. Vale a pena

salientar que a escolha das restrições é fundamental para a eficácia alcançada e que é importante a estruturação em diferentes níveis para redução de custos e facilitar a automatização.

Os resultados iniciais obtidos com o experimento piloto: número de casos de teste requeridos; aumento da eficácia; e aumento da cobertura obtida na ferramenta Proteum, motivaram a implementação de módulos para dar apoio à utilização dos critérios restritos, descritos no Capítulo 4. Os módulos são extensões propostas à ferramenta POKE-TOOL e permitiram a realização de um experimento de aplicação com programas mais complexos para verificar a aplicabilidade e o comportamento dos critérios restritos. Esse experimento é descrito no Capítulo 5.

Analisando-se as propriedades dos Critérios Baseados em Restrições, tem-se que a utilização dos Critérios Restritos, com um conjunto comum de restrições, contribui para preservar a relação de inclusão quando o fator eficácia é considerado. No Capítulo 6 é proposta uma estratégia incremental para a geração de dados de teste que utiliza os critérios de teste restritos e que preserva a hierarquia entre eles, com relação a eficácia. Ela tem o objetivo de facilitar a automatização da geração de dados de teste e reduzir os efeitos causados por caminhos não executáveis nessa atividade.

Capítulo 4

Aspectos de Implementação dos Critérios Restritos

Nesse capítulo é apresentada uma proposta de extensão da ferramenta POKE-TOOL com o objetivo de apoiar a utilização dos critérios estruturais restritos, definidos no capítulo anterior. Essa proposta visa a implementar módulos de apoio a geração de dados de teste. Esses módulos levam em consideração o aprendizado obtido durante os experimentos descritos no Capítulo 3, e em outros experimentos conduzidos por Vergilio, Maldonado e Jino [VMJ95b, VMJ96b, Ver92, VMJ96a]. Da experiência obtida, derivou-se um conjunto de diretrizes que auxiliam a etapa de geração de dados de teste para a satisfação de critérios, apresentadas com detalhe no Capítulo 6.

Primeiramente é dada a estrutura atual da ferramenta POKE-TOOL e em seguida a proposta de extensão. Parte dessa proposta foi implementada e utilizada num experimento de avaliação do critério todos-potenciais-usos restritos, descrito no Capítulo 5. Os principais aspectos considerados durante esta implementação são discutidos.

4.1 A Ferramenta POKE-TOOL

A ferramenta de teste POKE-TOOL [Cha91] foi implementada em linguagem C com o objetivo de tornar possível a utilização dos Critérios Potenciais Usos. São três os critérios básicos implementados pela POKE-TOOL: *critério todos-potenciais-usos*, *critério todos-potenciais-usos/du*, *critério todos-potenciais-du-caminhos*. Também é possível utilizar o *critério todos-nós* e o *critério todos-ramos*. O projeto da ferramenta POKE-TOOL pode ser visualizado na Figura 4.1.

A ferramenta é multi-linguagem e aceita programas escritos em diferentes linguagens de programação. O programa a ser testado, em seu código fonte, é traduzido para uma linguagem intermediária (LI) que está associada ao grafo de fluxo de controle do programa (GFC). Os módulos *li* e *nli* são responsáveis por essa tradução e são dependentes da linguagem do programa em teste. Eles já foram configurados para a linguagem C, Pascal, FORTRAN e COBOL [Cha91].

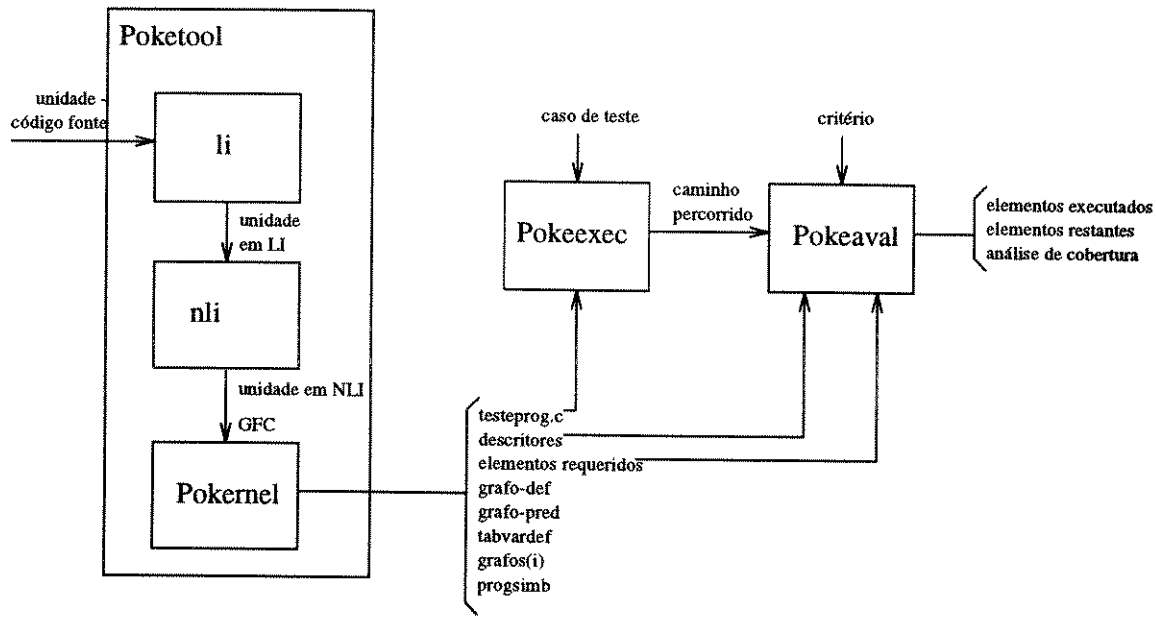


Figura 4.1: Ferramenta POKE-TOOL - Principais Módulos

A POKE-TOOL tem como entrada o programa a ser testado, o critério a ser utilizado e um conjunto de casos de teste. Na hipótese de este conjunto ser vazio a POKE-TOOL fornecerá um conjunto de elementos necessários para satisfazer o critério selecionado, orientando desta forma a seleção de casos de teste. Se o conjunto não for vazio, será produzida uma relação de elementos requeridos pelo critério mas ainda não cobertos, realizando assim a análise da adequação do conjunto de testes fornecido com relação ao critério selecionado.

O dados gerados pelos módulos li e nli são utilizados pelo módulo *Pokernel* que associa ao GFC as variáveis de programas definidas em cada nó e gera o *grafo/def.* O módulo *Pokernel* realiza uma análise do programa e gera um programa instrumentado, o *testeprog.c.* Esse programa contém "pontas de provas" associadas a cada nó do GFC e produz o caminho do GFC percorrido. O *Pokernel* gera toda informação estática referente aos Critérios Potenciais Usos. Ele é quem gera os *grafos(i)* que serão utilizados para gerar nós, arcos, associações e caminhos requeridos, bem como os descritores para cada um dos critérios implementados que auxiliarão na etapa de avaliação de casos de teste e no cálculo da cobertura. O *Pokernel* também gera as informações utilizadas pelas rotinas de tratamento de não executabilidade. Ele gera o *grafo/pred* que associa a cada nó do GFC o predicado do comando condicional correspondente quando o comando existir.

Informações para a execução simbólica do programa também são geradas pelo Pokernel que, juntamente com os grafos gerados são utilizados pelas rotinas de tratamento de não executabilidade.

O módulo *Pokeexec* tem, como entrada, os dados de teste para o programa *testeprog.c* e produz o caminho percorrido durante a sua execução. As entradas fornecidas e saídas obtidas

durante a execução do programa, são armazenadas pela POKE-TOOL. O módulo avaliador utiliza os descritores e os caminhos percorridos pelos casos de teste, para produzir, dado um critério, a lista de elementos requeridos que foram exercitados, e uma lista de elementos restantes, não cobertos pelos casos de teste.

4.2 Extensões Propostas à Ferramenta POKE-TOOL

A Figura 4.2 apresenta novos módulos que foram acrescentados ao projeto original da ferramenta: *Pokeheuris, Pokepadrao, Pokepaths, Cb-kernel, Cb-aval* com o objetivo de facilitar a geração de dados de teste e de implementar e avaliar os Critérios Potenciais Usos Restritos.

A seguir, são discutidas as funções dos módulos acrescentados e discutidos aspectos de implementaçao dos módulos *Cb-kernel* e *Cb-aval.* Os módulos *Pokeheuris, Pokepadrao* foram implementados no contexto de outros trabalhos [Ver92]. O módulo *Pokepaths* faz parte de um trabalho de mestrado e está sendo implementado. Aspectos de implementação desses módulos não serão descritos nessa dissertação. No Apêndice C são apresentadas as descrições dos "scripts" correspondentes a esses módulos e também uma descrição de como utilizá-los.

4.2.1 Módulos de Tratamento de Não Executabilidade

Os módulos de tratamento de não executabilidade têm a função de eliminar entre os elementos requeridos, os elementos não executáveis. As facilidades implementadas por esses módulos são: heurística de Frankl [Fra87] para determinar associações e padrões de não executabilidade e módulo de inserção/remoção de padrões.

O módulo *Pokepadrao* tem como entrada o critério ao qual o padrão deve ser aplicado e um padrão de não executabilidade. O módulo reconhece que tipo de padrão é fornecido e checa a sua conformidade com o critério dado; além disso, determina a quais outros critérios o padrão serve. Os elementos, que incluem o padrão dado, são retirados do arquivo de elementos requeridos restantes e colocados num arquivo de elementos não executáveis. O módulo também mantém um registro de padrões, o que possibilita inserir, remover, habilitar e desabilitar padrões durante a avaliação dos casos de teste.

O módulo *Pokeheuris* implementa a heurística de Frankl,. descrita no Capítulo 2. Utilizando as informações simbólicas, o *grafo-def/pred* e os *grafos(i)* gerados pelo *Pokernel,* é realizada a avaliação simbólica de predicados dos caminhos candidatos a cobrir uma dada associação. A heurística poderá decidir em alguns casos pela não executabilidade dos caminhos e conseqüentemente da associação. Em outros casos, nada poderá ser decidido. Uma associação não executável é um padrão de não executabilidade que deve ser passado ao módulo *Pokepadrao* para a eliminação dos elementos correspondentes.

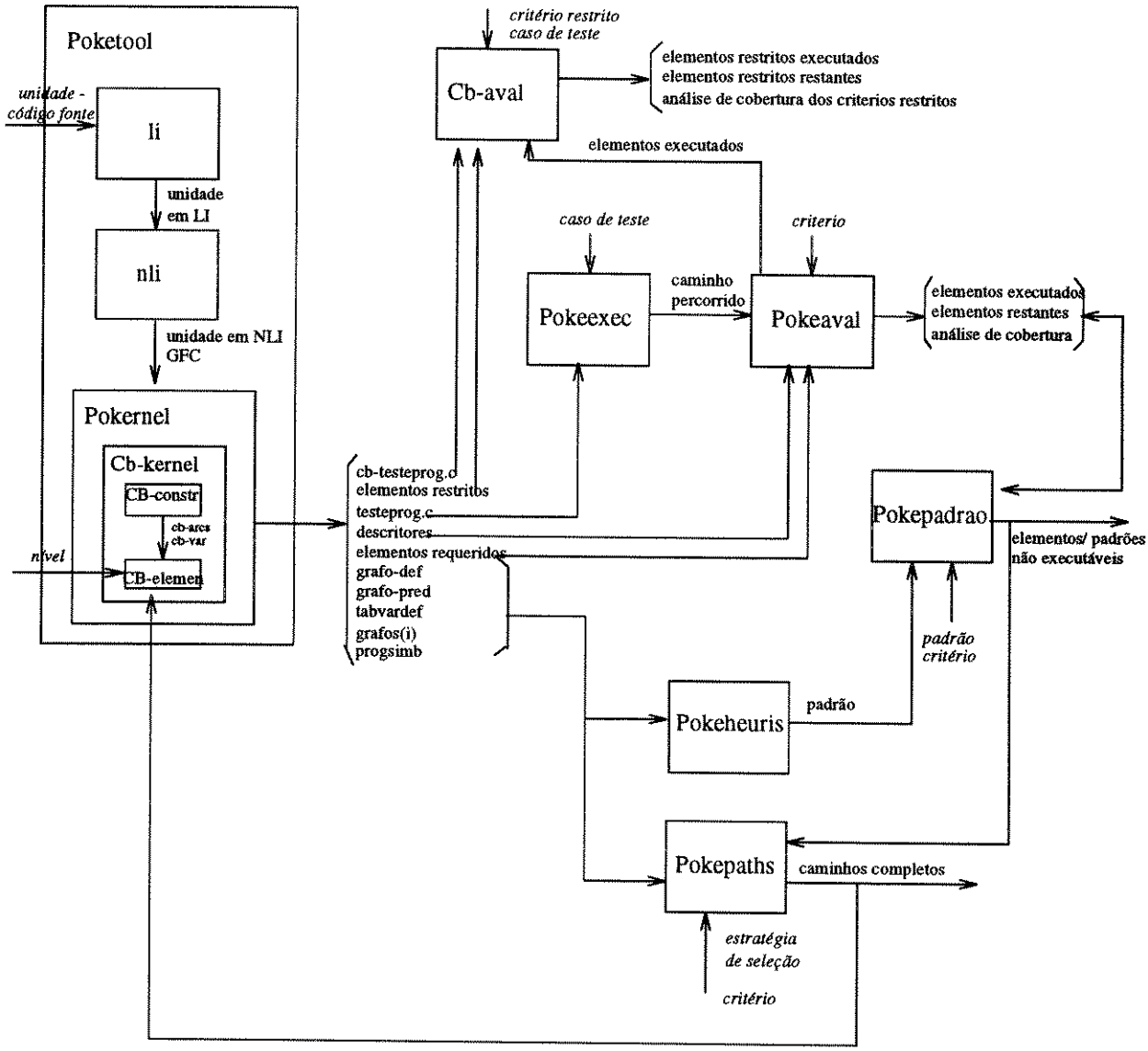


Figura 4.2: Ferramenta POKE-TOOL - Extensão

Os módulos descritos foram incorporadas ao ambiente POKE-TOOL e as rotinas de tratamento de não executabilidade foram implementadas e testadas na versão PC-DOS da ferramenta [Ver92]. Atualmente estão sendo implementados "scripts" que ativam ou não esses módulos mediante opção do usuário [BJVM97] para a versão SUN-UNIX da POKE-TOOL (Ver descrição desses "scripts" no Apêndice C).

4.2.2 Módulo para Geração de Caminhos Completos

Esse módulo, chamado *Pokepaths*, tem o objetivo de gerar uma lista de caminhos completos para cobrir um dado elemento ou uma lista de elementos requeridos, por um dado critério. O módulo tem como entrada uma estratégia a ser utilizada para a seleção dos caminhos. Essa estratégia pode considerar vários aspectos do programa ou caminho: número de predicados; número de computações; número de definições e/ou usos de variáveis; complexidade das estruturas de dados manipuladas ao longo do caminho; testabilidade, ou seja, o caminho é ou não freqüentemente executado durante a utilização do programa; complexidade das estruturas de controle envolvidas; facilidade de automatização, ou seja, caminhos cujas as condições de execução sejam lineares; escolha aleatória, etc. Para selecionar o conjunto de caminhos completos leva-se em consideração os padrões de não executabilidade disponíveis, a fim de diminuir o número de caminhos não executáveis entre os escolhidos.

Numa primeira etapa está sendo implementada a estratégia que considera o número de predicados ao longo do caminho. A idéia é poder avaliar a influência do número de predicados de um caminho, na sua executabilidade. Para isso, foi planejado um experimento de avaliação da estratégia. Maiores detalhes de implementação e resultados obtidos são descritos por Peres [PVJM97] [Per97].

4.2.3 Módulos de Apoio à Utilização de Critérios Restritos

São dois os módulos básicos para a utilização dos Critérios Restritos. O primeiro é o módulo chamado *Cb-kernel*, que é formado pelos módulos: o *Cb-constr*, que gera as restrições a serem associadas aos elementos requeridos; e *Cb-elem*, módulo que gera os elementos requeridos a partir dessas restrições. O *Cb-kernel* faz parte do *Pokernel* e é ativado mediante opção do usuário. O segundo módulo, o *Cb-aval*, é o que faz a avaliação dos casos de teste segundo os critérios restritos.

O módulo gerador de restrições, *Cb-constr*, utiliza informações tais como GFC (grafo de fluxo de controle do programa), *grafo-def/pred*, *tabvardef* (tabela de variáveis definidas em cada nó). As restrições são estabelecidas na linguagem do programa em teste. É desejável, para facilitar a geração automática de dados, que elas sejam escritas em função das variáveis de entrada do programa, no entanto, essa é uma tarefa muito complexa pois envolve técnicas de execução simbólica

global de programas. Portanto, numa primeira etapa de implementação, as restrições envolvem quaisquer variáveis de programa.

As restrições, quando estabelecidas, estão associadas às variáveis de programa e aos arcos do grafo de fluxo de controle; estas informações estão representadas respectivamente em *cb-var* e *cb-arcs*. Um programa chamado *cb-testeprog.c* é gerado pelo *Cb-constr*, ele contém pontos de inserção de restrições. Cada restrição gerada em *cb-var* ou *cb-arcs* está associada a um ponto de inserção, a ser utilizado durante a avaliação dos critério restritos.

O módulo *Cb-elemen* gera uma lista dos elementos requeridos pelos critérios restritos determinada pela opção *nível*. Dependendo do nível escolhido, outras informações serão necessárias, tais como caminhos candidatos a cobrir o elemento, etc.

O módulo *Cb-aval* faz a análise da adequação do critério restrito selecionado com relação a um conjunto de casos de teste. O módulo avaliador *Pokeaval* fornece a lista de elementos cobertos pelo conjunto, o módulo *Cb-aval* insere a restrição associada a cada elemento nos pontos de inserção de *cb-testeprog.c*. O programa *cb-testeprog.c* é executado pelo menos até o ponto de inserção para verificar se a restrição foi satisfeita, caso no qual o elemento restrito foi coberto. O módulo *Cb-aval* fornece uma lista de elementos restritos cobertos pelos casos de teste e uma lista de elementos restantes, a serem cobertos, bem como a porcentagem de cobertura para os critérios restritos.

Um próximo passo no sentido de automatizar as atividades de geração de dados de teste é utilizar os padrões de não executabilidade disponíveis para eliminar os elementos restritos não executáveis, e ainda, implementar módulos para gerar dados de teste automaticamente. Os dados de teste seriam as soluções para o sistema dado pelas condições dos caminhos completos gerados por *Pokepaths*, e pelas restrições requeridas pelos critérios restritos, que têm alta probabilidade de revelar erros.

4.2.4 Utilização das Novas Funcionalidades

Nesta seção será fornecida uma descrição de como as novas funcionalidades propostas poderão ser utilizadas. Primeiramente executa-se o módulo *Poketool*, fornecendo o programa que será testado. Serão geradas todas as informações estáticas necessárias aos outros módulos, entre essas, os elementos requeridos pelos critérios Potenciais-Usos e Potenciais-Usos Restritos.

O módulo *Pokeexec* é ativado sempre que se desejar executar um dado de teste. O dado de teste é armazenado bem como a saída obtida e o caminho percorrido.

O módulo *Pokeaval* é chamado para avaliar a adequação de um conjunto de casos de teste com relação aos critérios Potenciais-Usos. O módulo *Cb-aval* utiliza o módulo *Pokeaval* e faz a avaliação com relação aos critérios Potenciais-Usos Restritos, executando o programa *cb-testeprog.c* para verificar se as restrições foram ou não satisfeitas. Se nenhum caso de teste tiver sido executado,

ambos os avaliadores fornecerão a lista de elementos requeridos pelos critérios a serem cobertos.

O módulo *Pokepadrao* é ativado sempre que um padrão de não executabilidade for identificado pelo usuário. Além disso, o usuário poderá a qualquer momento inserir, remover, habilitar e desabilitar padrões. O módulo *Pokepadrao* provocará alterações nos arquivos de elementos executados e/ou elementos não executados gerados. Por exemplo, se alguma avaliação já tiver sido feita, os elementos não executáveis correspondentes aos padrões inseridos serão removidos do arquivo de elementos restantes e uma nova cobertura é calculada.

O módulo *Pokeheuris*, para determinar associações não executáveis, pode ser ativado antes ou depois da execução/avaliação de casos de teste. O módulo poderá através da heurística de Frankl determinar a não executabilidade de uma associação ou uma lista de associações fornecidas. Uma associação não executável é um padrão de não executabilidade. Quando uma associação não executável é identificada pelo *Pokeheuris* o módulo *Pokepadrao* é ativado.

O módulo *Pokepaths* orienta a geração de dados de teste, e pode ser ativado a qualquer momento para fornecer um conjunto de caminhos que cubram um elemento ou uma lista de elementos requeridos. De posse desses caminhos, o usuário necessita apenas gerar dados de teste para executar o conjunto fornecido.

4.3 Aspectos de Uma Implementação dos Módulos *Cb-kernel* e *Cb-aval*

A fim de conduzir um experimento de avaliação do critério todos-potenciais-usos-restritos descrito no próximo capítulo, uma versão do módulo *Cb-aval* foi implementada. Nesta etapa de implementação, o único critério disponível é o critério todos-potenciais-usos-restritos, também considera-se um único nível para as restrições. As informações que estão disponíveis são os predicados associados aos nós. Manualmente, é aplicada a técnica BRO e são derivadas as restrições associadas a cada arco do GFC, dadas pelo arquivo *cb-arcs*. Os conjuntos de restrições St e Sf derivados pela técnica são associados ao arco que corresponde respectivamente aos resultados True e False da avaliação do predicado. Durante a determinação das restrições para predicados compostos pode ser necessário escolher um elemento do conjunto St ou Sf de um dos sub-predicados, optou-se sempre pela escolha aleatória de um elemento, nenhum outro critério de escolha foi adotado.

Um "script" é criado para gerar os elementos requeridos para o critério todos-potenciais-usos restritos a partir das informações geradas manualmente. Esse "script" é uma versão inicial do módulo *Cb-elemen*. Os scripts correspondentes ao *Cb-aval* e a versão inicial de *Cb-elemen* estão no Apêndice C.

O programa *cb-testeprog.c* também é gerado manualmente. Ele é gerado a partir do *programa.nli* e pontos de inserção de restrições são colocadas no programa original. Para a sua geração

deve-se seguir um modelo de instrumentação que determina onde inserir pontos de inserção para cada comando básico da LI, e como tratar chamadas de funções e outras operações da linguagem C.

As principais características desse modelo são apresentadas na Figura 4.3. Na figura estão representados os pontos de inserção das restrições de St e Sf para os comandos *if*, *for*, *while* e *repeat*. O comando *case* não foi instrumentado, pois a restrição $St = \{=\}$ é sempre satisfeita quando o arco é executado, para todas as alternativas do case. Se existe um arco "default", o conjunto Sf é dado por todas as combinações de $>$ e $<$ para todas as alternativas do case. Isso gera um número muito grande de restrições, onde a maioria é não executável, por envolver a mesma variável (exemplo $x > 5 \&\& x < 0$ para apenas duas alternativas). Para o comando *repeat* é adicionada uma variável booleana v_repeat que armazena o valor da condição para uniformizar o teste da restrição após o teste do predicado.

O teste da restrição é realizado imediatamente após a avaliação do predicado para que nenhuma operação seja realizada mais de uma vez, sendo a execução do programa interrompida apenas quando a restrição for satisfeita. Por exemplo, considere dois laços aninhados e uma restrição St do laço mais interno que pode ser satisfeita somente na terceira interação do laço mais externo. Se a execução do programa for interrompida quando a restrição não é satisfeita o elemento restrito não estará sendo coberto pelo caso de teste como deveria. Portanto, quando uma restrição não é satisfeita, existe a necessidade de se manter o estado do programa após o teste de uma restrição e de se executar o programa até o final. Quando são realizadas, dentro do predicado, algumas operações que alteram o estado do programa, as restrições deverão desconsiderá-las.

A Tabela 4.1 mostra como alguns comandos da linguagem C são convertidos em restrições. Vale ressaltar que nem as atribuições e nem as chamadas de funções são realizadas duas vezes. Quando uma atribuição é feita após a comparação, a restrição é elaborada de maneira a não ser afetada por esta atribuição. Numa chamada de função, uma variável de mesmo tipo de retorno da função é utilizada na restrição, e o predicado é ligeiramente transformado. Variáveis do tipo *pointer* nunca serão menores que o valor $NULL$, do tipo *char* nunca menores que EOF ou 0, e portanto essas restrições não executáveis não devem ser geradas.

O módulo *Cb-aval* tem como entrada os arquivos *cb-arcs.tes*, gerados manualmente, onde estão as associações do tipo BRO associadas aos arcos primitivos. A partir desse arquivo o script *geracbpu*, já implementado, que faz parte do módulo *Cb-kernel*, gerará o arquivo *cb-puassoc.tes* que contém as associações restritas requeridas pelo critério todos-potenciais-usos restritos e o arquivo *cb-relpu*, que relaciona cada restrição com o seu ponto de inserção em *cb-testeprog.c*.

No Apêndice C encontram-se disponíveis os arquivos *cb-testeprog.c*, *cb-arcs.tes*, *cb-assoc.tes*, que são entradas para o *Cb-aval* e os arquivos por ele gerados *cb-puoutput.tes*, *cb-puexe.tes*. O programa utilizado é o programa *max* do Capítulo 3.

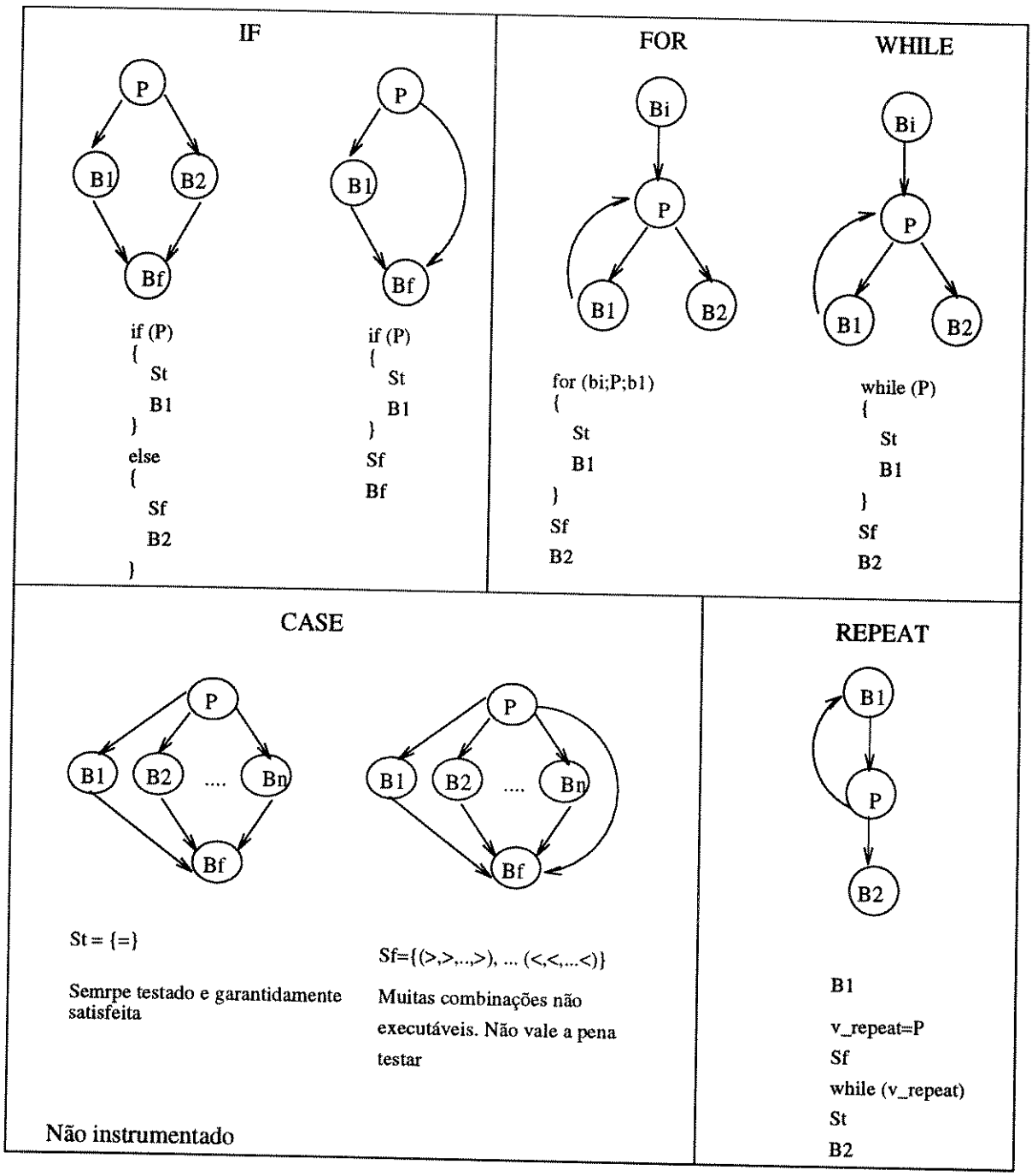


Figura 4.3: Modelo para Inserção de Restrições para o *Cb-kernel*

Tabela 4.1: Convertendo Comandos da Linguagem C em Restrições

Predicado	Restrições
atribuição de variáveis no predicados	
$((c = x) < 4)$	$c == 4, c > 4, c < 4$
$(++a < 0)$	$a == 0, a > 0, a < 0$
$(a-- < 0)$	$a+1 == 0, a+1 > 0, a+1 < 0$
chamada de funções	
$((c = getchar()) <' a')$	$c ==' a', c >' a', c <' a'$
$(f() < 0) \Rightarrow (v\_f = f() < 0)$	$v\_f == 0, v\_f > 0, v\_f < 0$
variáveis do tipo pointer	
$(p == NULL)$	$p == NULL, p > NULL$
variáveis do tipo char	
$(c == 0)$	$c == 0, c > 0$
$(c == EOF)$	$c == EOF, c > EOF$
constante numérica	
(1)	

4.4 Considerações Finais

A extensão proposta à ferramenta POKE-TOOL visa a utilização dos Critérios Baseados em Restrições. O módulo *Cb-aval* foi implementado para realizar um experimento de aplicação desses critérios. Com isso, o modelo de instrumentação para inserção das restrições associadas aos arcos pôde ser validado. O *Cb-kernel*, composto dos módulos *Cb-elem* e *Cb-restr*, ainda precisa ser implementado; muitas das informações necessárias para sua implementação estão diponíveis e são geradas pelo *Pokernel*.

As rotinas de tratamento de não executabilidade que estão sendo incorporadas à versão UNIX da POKE-TOOL e o módulo *Pokepaths* permitem que a POKE-TOOL forneça um suporte às atividades de geração de dados de teste como o auxílio à determinação de elementos não executáveis, e determinação, segundo diferentes estratégias, de conjuntos de caminhos completos para cobrir os elementos executáveis.

Pretende-se num segundo passo, com ajuda de técnicas de execução simbólica ou dinâmica, gerar as restrições requeridas pelos critérios restritos em função das variáveis de entrada do programa, e determinar as condições dos caminhos completos gerados, e assim, poder gerar dados de teste automaticamente, para a satisfação dos critérios disponíveis na POKE-TOOL, com o auxílio de programas que resolvam o conjunto de condições formado.

Motivados pelos resultados do projeto piloto descrito no Capítulo 3, e utilizando o módulo *Cb-aval* implementado foi realizado um experimento de avaliação do critério todos-potenciais-usos-restritos descrito no Capítulo 5.

Capítulo 5

Experimento de Avaliação

Esse capítulo descreve um experimento de avaliação do critério todos-potenciais-usos restritos. São apresentados: principais objetivos, como foi realizada a coleta de dados e uma análise dos principais resultados obtidos. Os critérios foram aplicados a programas mais complexos do que os utilizados no experimento piloto descrito no Capítulo 3. Foram utilizadas as versões iniciais dos módulos *Cb-kernel* e *Cb-aval*, incorporados ao ambiente POKE-TOOL e cujos aspectos de implementação foram discutidos no capítulo anterior. O experimento foi conduzido com os seguintes objetivos:

- estudar a aplicabilidade de critérios restritos em programas reais, verificando: dificuldade de satisfação, número de casos de teste necessários e eficácia desses critérios.
- comparar duas estratégias de geração de dados de teste para satisfazer os critérios: uma estratégia aleatória e uma estratégia "ad-hoc". A estratégia "ad-hoc" consistiu em utilizar aspectos funcionais e estruturais do programa em teste sem aplicar nenhuma técnica em particular.
- comparar o critério todos potenciais-usos-restritos e seu correspondente critério estrutural todos potenciais-usos em termos de eficácia e número de casos de teste.
- estudar as principais causas de não executabilidade encontradas durante a aplicação do critério todos potenciais-usos-restritos, ou seja, quais tipos de restrições mais geraram elementos não executáveis, e quais as causas de inconsistência, com as condições dos caminhos que cobrem esses elementos.

5.1 Descrição do Experimento e Coleta dos Resultados

O experimento foi conduzido utilizando nove programas UNIX, que são de domínio público: *cal*, *checkeq*, *comm*, *col*, *crypt*, *look*, *spline*, *tr*, *uniq*, utilizado em dois outros experimentos com

Tabela 5.1: Principais Características dos Programas Utilizados

Progr.	No.Unidades	No.Linhas Código	Complex. McCabe	No.Nós ≠	No.Pred.	No.Pred. Compostos (%)
cal	4	203	28	59	20	4
checkeq	2	104	26	61	24	5
col	7	301	65	118	37	5
comm	6	170	40	84	24	3
crypt	3	132	21	52	18	1
look	4	146	33	71	17	3
spline	7	332	64	119	40	7
tr	3	141	44	97	32	4
uniq	6	140	34	67	21	3
Total	40	1669	355	728	239	35 (14,64%)

análise de mutantes [WMM94]. Foram duas as razões para a escolha desse conjunto: 1) já estavam disponíveis um conjunto de dados gerados aleatoriamente, e um conjunto de versões incorretas para cada um dos programas; 2) possibilitar futuras comparações com o critério análise de mutantes.

Cada um dos programas é constituído de um conjunto de unidades. Algumas características sobre cada programa, dadas pela somátoria das características de cada unidade, foram coletadas e são apresentadas na Tabela 5.1. Existem diferentes versões incorretas para cada programa, geradas por Wong [WMM94]. Os erros introduzidos são alterações simples e correspondem a pequenas mutações da versão correta e são fáceis de serem revelados. Um dos objetivos do experimento é verificar o aumento da eficácia utilizando-se o critério todos-potenciais-usos-restritos. No entanto, o aumento esperado é pequeno, pois, o critério todos-potenciais-usos é um critério forte e revela a maioria dos erros introduzidos nos programas.

O experimento consistiu dos seguintes passos, realizados para cada programa:

1. **Geração dos Elementos Requeridos** pelos critérios todos-potenciais-usos e todos-potenciais-usos-restritos.

   - Geração dos elementos requeridos pelo critério todos potenciais-usos, para cada unidade, utilizando a ferramenta POKE-TOOL.
   - Geração manual dos arquivos *cb-arcs* para cada unidade e o programa instrumentado *cb-testeprog.c* a ser utilizado pelo *Cb-aval* (especificados no capítulo anterior). Os *cb-arcs* gerados contêm as restrições do tipo BRO, para os predicados do programa, associadas aos respectivos arcos do grafo de fluxo de controle. Para simplificar, na geração das restrições foi considerado apenas o nível 3, dentre especificados no Capítulo 3, apenas a técnica BRO foi aplicada.

- Geração dos elementos restritos a partir do arquivo *cb-arcs*: associar a cada elemento $(i,(j,k),x)$, as restrições derivadas no passo anterior para o arco (j,k). Esse passo é feito automaticamente ativando-se o script *Cb-elemen*.

2. A **Geração "Ad-hoc" de dados de teste** foi realizada para satisfazer ambos os critérios; para gerar os dados de teste, fez-se uso da especificação do programa, do código fonte e do grafo de fluxo de controle, sem aplicar nenhuma técnica de geração. Toda a informação disponível foi utilizada para cobrir os elementos requeridos.

   - Geração de dados para o critério todos-potenciais-usos e submissão dos dados de teste à ferramenta POKE-TOOL e obtenção da cobertura com relação a esse critério. Nesse passo também foram identificados os elementos não executáveis. Ao final, obteve-se um conjunto de casos de teste adequado ao critério todos-potenciais-usos
   - Geração de dados adicionais para satisfazer o critério todos potenciais-usos restritos, e submissão dos dados de teste ao módulo *Cb-aval* para obtenção da cobertura com relação a esse critério. Também aqui, foram identificados os elementos restritos não executáveis e anotadas as principais causas de não executabilidade encontradas. Ao final, obteve-se um conjunto de dados de teste adequado ao critério todos-potenciais-usos-restritos.

   A Tabela 5.2 apresenta os resultados da análise de adequação dos dados "ad-hoc" gerados: o número de elementos requeridos, a porcentagem de cobertura, número de elementos não executáveis encontrados e o número de casos de teste necessários para os critérios PU (todos potenciais-usos) e PU-R (todos potenciais-usos-restritos).

3. **Obtenção da cobertura dos dados de teste gerados aleatoriamente com relação a ambos os critérios.** Não foi necessária realizar a geração aleatória de dados de teste, foi utilizado o mesmo conjunto de dados utilizados por Wong [WMM94]. Esse conjunto foi submetido a ferramenta POKE-TOOL para obtenção da cobertura do critério todos-potenciais-usos. Foi gerada uma lista dos dados que realmente contribuiram para a cobertura obtida. O mesmo conjunto foi utilizado para determinar a cobertura do critério todos-potenciais-usos-restritos. O módulo *Cb-aval* também produz uma lista com os dados de teste adequados a esse critério. Dados de teste adicionais não foram gerados, de forma que existem elementos executáveis entre os não executados.

   A Tabela 5.3 apresenta os resultados sobre a análise da adequação dos casos de teste gerados aleatoriamente com relação aos critérios PU (todos potenciais-usos) e PU-R (todos potenciais-usos-restritos). A cobertura apresentada na Tabela 5.2 é a cobertura real, visto que os elementos restantes são todos não executáveis, ou seja, conjuntos adequados a ambos os critérios foram gerados de maneira "ad-hoc". Isso não acontece com a cobertura apresentada na Tabela 5.3. Existem elementos executáveis entre os elementos não cobertos.

Tabela 5.2: Cobertura para cada Programa - Geração "Ad-Hoc"

Programa	Crit.	Elem. Req.	Elem. Exe.	Elem. N.Exe.	Cobertura %	No.Casos Utiliz.
cal	PU	242	173	69	71,49	12
	PU-R	488	308	180	63,11	17
checkeq	PU	582	468	114	80,41	76
	PU-R	1539	1068	471	69,40	164
col	PU	999	874	125	87,49	69
	PU-R	1807	1405	402	77,75	75
comm	PU	427	277	150	64,87	68
	PU-R	884	501	383	56,67	74
crypt	PU	322	257	65	79,81	14
	PU-R	650	455	195	70,00	19
look	PU	524	437	87	83,00	34
	PU-R	1045	798	247	76,36	40
spline	PU	999	793	206	79,38	57
	PU-R	2352	1396	956	59,35	63
tr	PU	1527	594	933	38,90	30
	PU-R	3040	1026	2014	33,75	35
uniq	PU	262	230	32	87,79	36
	PU-R	552	416	136	75,36	42
TotalPU		5884	4103	1781	69,73	396
TotalPU-R		12357	7373	4984	59,67	529

4. **Análise da Eficácia** dos 4 conjuntos de casos de teste obtidos nos passos anteriores. Nesse passo foram utilizadas as versões incorretas de cada programa, geradas por Wong [WMM94].

   - Classificação dos tipos de erros existentes em cada versão. Os erros foram classificados em erros de domínio, erros de computação e erros em estrutura de dados. A Tabela 5.4 apresenta o total de versões incorretas para cada programa em cada categoria. Note que, 58.76% dos programas incorretos apresentam erros de computação.
   - Geração de tabelas de eficácia para os 4 conjuntos de dados de teste, através da execução das versões incorretas e comparação com as saídas dos programas originais.

As Tabelas 5.5, 5.6, 5.7 e 5.8 apresentam os resultados da análise da eficácia dos dados de teste que contribuíram para a cobertura do critério todos-potenciais-usos e todos-potenciais-usos restritos, utilizando geração "ad-hoc" e geração aleatória, respectivamente. Note que, nessas tabelas existem duas entradas para cada programa. A primeira entrada mostra o número de erros (e porcentagem) revelados em cada categoria. A segunda entrada mostra o número de erros (e porcentagem) não revelados.

Tabela 5.3: Cobertura para cada Programa - Geração Aleatória

Programa	Crit.	Elem. Req.	Elem. Exe.	Elem. N.Exe.	Cobertura %	No.Casos Utiliz.
cal	PU	242	173	69	71,49	7
	PU-R	488	309	179	63,32	17
checkeq	PU	582	345	237	59,28	13
	PU-R	1539	656	883	42,63	13
col	PU	999	633	366	63,36	23
	PU-R	1807	982	825	54,34	24
comm	PU	427	258	169	60,42	58
	PU-R	884	464	420	52,49	59
crypt	PU	322	220	102	68,32	4
	PU-R	650	381	269	58,62	4
look	PU	524	387	137	73,85	29
	PU-R	1045	710	335	68,00	29
spline	PU	999	659	340	65,97	28
	PU-R	2352	1186	1166	50,43	36
tr	PU	1527	365	1162	23,90	9
	PU-R	3040	627	2413	20,62	11
uniq	PU	262	215	47	82,06	31
	PU-R	552	376	176	66,90	32
TotalPU		5884	3255	2629	55,32	202
TotalPU-R		12357	5691	6666	46,05	225

Tabela 5.4: Classificação dos Erros Introduzidos em Cada Programa

Prgrama	Erro Comp.	Erro Dom.	Erro Est.Dad	Total
cal	12	7	1	20
checkeq	13	9		22
col	25	10	2	37
comm	8	11		19
crypt	13	4	1	18
look	10	9	1	20
spline	11	9		20
tr	13	6		19
uniq	9	9	1	19
Total	114 (58,76%)	74 (38,14%)	6 (3,09%)	194

Tabela 5.5: Eficácia - PU-Geração "Ad-Hoc"

Prg	Erro Comp.	Erro Dom.	Erro Estrutura Dados	Total
cal	12 (100%)	6 (85.71%)	1 (100%)	19 (95%)
		1 (14,29%)		1 (5%)
checkeq	12 (92,31%)	8 (88,89%)		20 (90,91%)
	1 (7,69%)	1 (11,11%)		2 (9,09%)
col	25 (100%)	10 (100%)	2 (100%)	37(100%)
comm	8 (100%)	11 (100%)		19(100%)
crypt	12 (92,31%)	3 (75%)		15 (83,33%)
	1 (7,69%)	1 (25%)	1 (100%)	3 (16,67%)
look	10 (100%)	9 (100%)	1(100%)	20(100%)
spline	11 (100%)	5 (56%)		16 (80%)
		4 (44%)		4 (20%)
tr	13 (100%)	6 (100%)		19 (100%)
uniq	9 (100%)	9 (100%)		18 (94,74%)
			1 (100%)	1 (5,26%)
totrev	112 (98,25%)	67 (91,54%)	4 (66,67%)	183 (94,33%)
totnrev	2 (1,75%)	7 (9,46%)	2 (33,33%)	11 (5,67%)
tot	114	74	6	194

Tabela 5.6: Eficácia - PU-R - Geração "Ad-Hoc"

Prg	Erro Comp.	Erro Dom.	Erro Estrutura Dados	Total
cal	12 (100%)	7 (100%)	1 (100%)	20(100%)
checkeq	12 (92,31%)	8 (88,89%)		20 (90,91%)
	1 (7,69%)	1 (11,11%)		2 (9,09%)
col	25 (100%)	10 (100%)	2 (100%)	37 (100%)
comm	8 (100%)	11 (100%)		19 (100%)
crypt	12 (92,31%)	4 (100%)		16 (88,89%)
	1 (7,69%)		1 (100%)	2 (11,11%)
look	10 (100%)	9 (100%)	1 (100%)	20 (100%)
spline	11 (100%)	5 (56%)		16 (80%)
		4 (44%)		4 (20%)
tr	13 (100%)	6 (100%)		19 (100%)
uniq	9 (100%)	9 (100%)		18 (94,74%)
			1 (100%)	1 (5,26%)
totrev	112 (98,25%)	69 (93,24%)	4 (66,67%)	185 (95,36%)
totnrev	2 (1,75%)	5 (6,73%)	2 (33,33%)	9 (4,64%)
tot	114	74	6	194

Tabela 5.7: Eficácia - PU-Geração Aleatória

Prg	Erro Comp.	Erro Dom.	Erro Estrutura Dados	Total
cal	10 (83,33 %)	7 (100%)	1 (100%)	18 (90%)
	2 (16,67%)			2 (10%)
checkeq	12 (92,31%)	7 (77,78%)		19 (86,36%)
	1 (7,69%)	2 (22,22%)		3 (13,64%)
col	25 (100%)	10 (100%)	2 (100%)	37 (100%)
comm	8 (100%)	11 (100%)		19 (100%)
crypt	12 (92,31%)	3 (75%)		15 (83,33%)
	1 (7,69%)	1 (25%)	1 (100%)	3 (16,67%)
look	10 (100%)	9 (100%)	1(100%)	20 (100%)
spline	10 (91%)	3 (33%)		13 (65%)
	1 (9%)	6 (67%)		7 (35%)
tr	13 (100%)	6 (100%)		19 (100%)
uniq	8 (88,89%)	9 (100%)		17 (89,47%)
	1 (11,11%)		1 (100%)	2 (5,26%)
totrev	108 (94,74%)	65 (87,84%)	4 (66,67%)	177 (91,24%)
totnrev	6 (5,26%)	9 (12,16%)	2 (33,33%)	17 (8,76%)
tot	114	74	6	194

Tabela 5.8: Eficácia - PU-R - Geração Aleatória

Prg	Erro Comp.	Erro Dom.	Erro Estrutura Dados	Total
cal	11 (91,67%)	7 (100%)	1 (100%)	19 (95%)
	1 (8,33%)			1 (5%)
checkeq	12 (92,31%)	7 (77,78%)		19 (86,36%)
	1 (7,69%)	2 (22,22%)		3 (13,64%)
col	25 (100%)	10 (100%)	2 (100%)	37 (100%)
comm	8 (100%)	11 (100%)		19 (100%)
crypt	12 (92,31%)	3 (%)		15 (83,33%)
	1 (7,69%)	1 (25%)	1 (100%)	3 (16,67%)
look	10 (100%)	9 (100%)	1(100%)	20 (100%)
spline	10 (91%)	3 (33%)		13 (65%)
	1 (9%)	6 (67%)		7 (35%)
tr	13 (100%)	6 (100%)		19 (100%)
uniq	8 (88,89%)	9 (100%)		17 (89,47%)
	1 (11,11%)		1 (100%)	2 (5,26%)
totrev	109 (95,61%)	65 (87,84%)	4 (66,67%)	178 (91,75%)
totnrev	5 (4,39%)	9 (12,16%)	2 (33,33%)	16 (8,25%)
tot	114	74	6	194

Maiores detalhes sobre a condução do experimento, tais como: tabelas para cada unidade de cada programa, tabelas de eficácia de cada caso de teste, dificuldades e problemas encontrados durante o experimento, podem ser obtidas em [VMJ97].

5.2 Análise dos Resultados

Nesta seção os resultados apresentados na seção anterior são analisados de acordo com os objetivos do experimento, estabelecidos no início deste capítulo. Primeiramente, serão discutidos algumas características dos programas estudados que influenciaram nos resultados. Note na Tabela 5.1 que os programas *col* e *spline* são os mais complexos segundo as medidas Kloc e de McCabe. Por isso, maior foi a dificuldade encontrada para geração de dados de teste e entendimento desses programas, e maior foi o número de elementos requeridos e casos de teste necessários.

Mais dois outros programas precisam ser mencionados: *checkeq* e *tr*. O programa *checkeq* apesar de possuir o menor número de linhas de código, e a segunda menor complexidade, requereu um número maior de associações que outros cinco programa,s e o maior número de casos de teste. Isso se deve à estrutura desse programa; ele apresenta um grande número de definições de variáveis, seis estruturas de controle diferentes, aninhadas de maneira a gerar um maior número de associações, um alto número de predicados compostos.

O mesmo não aconteceu com a rotina *tr* que, apesar de complexidade média, requereu o mais alto número de elementos, devido as mesmas razões ocorridas com *checkeq*, mas a porcentagem de elementos executáveis é a mais baixa entre todos os programas, e por isso, o número de casos de teste necessário foi o terceiro menor. Um outro ponto interessante é que apesar do alto número de elementos requeridos, a geração de dados de teste e determinação de não executabilidade dessa rotina não foi muito difícil, pois a maioria dos elementos são não executáveis devido a uma única causa.

5.2.1 Aplicabilidade dos Critérios Restritos

A condução do experimento confirmou a viabilidade de uso dos critérios restritos. Note na Tabela 5.2 que o número de casos de teste necessários não apresenta um crescimento proporcional ao número de elementos requeridos. Embora, tivesse sido requerido 110% mais de elementos pelo critério PU-R, o número de casos de teste necessários aumentou apenas 30,10% maior.

Novamente, os programas *checkeq* e *tr* destacam-se dos demais. Para *checkeq*, o critério PU-R requer 164% mais elementos que requer PU e 115,79% a mais casos de teste. Isso se deve ao tipo de restrições geradas (BRO). O programa *checkeq* possui a mais alta porcentagem de predicados compostos. Um maior número de restrições é gerado na presença de predicados compostos.

Para o programa *tr*, ao contrário, apesar de PU-R requerer 99% mais elementos que o critério PU, muitos são não executáveis devido as características da rotina. Portanto, para *tr* foi encontrada a menor diferença de cobertura (5,15%) entre os dois critérios e um aumento de 16,67% no número de casos de teste.

O mesmo testador foi utilizado para gerar dados adicionais para o critério PU-R. Notou-se que uma vez gerados os dados e identificados os elementos não executáveis para o critério PU, pouco esforço adicional foi necessário para identificar os elementos-restritos não executáveis e satisfazer as restrições requeridas pelo critério PU-R. Isso porque o esforço maior para realizar esses dois passos é gasto no entendimento do programa.

Outra questão quanto à aplicabilidade dos critérios restritos diz respeito ao tempo gasto com a avaliação. O tempo gasto na avaliação do critério PU-R é em média três vezes maior que o tempo gasto na avaliação do critério PU.

5.2.2 Estratégia Aleatória X Estratégia "Ad-Hoc"

A análise feita a seguir é resultado da comparação entre as estratégias de geração de dados de teste "ad-hoc" e aleatória, tais como descritas na Seção 2 deste capítulo.

Na utilização dos dados aleatórios não foram gerados dados de teste adicionais para cobrir os elementos executáveis que não puderam ser cobertos pelo conjunto de testes. Portanto, a cobertura obtida com esses dados foi menor. A diferença média de cobertura obtida é 14,41% para o critério PU e 13,17% para o critério PU-R. Mas, foram necessários na geração "ad-hoc", 96,04% a mais de casos de teste para o PU e 148,44% a mais para o PU-R. Isso pode ser explicado pelo fato de que alguns elementos requeridos tanto para o PU, e mais ainda para o PU-R, serem dificilmente executados, isto é, poucos casos de teste (às vezes apenas um) cobrem esses elementos e em geral esses dados não são gerados aleatoriamente.

O função *main* do programa, exemplifica o fato descrito acima. Pode ser visto na Tabela 5.9 que são requeridos 396 elementos para o PU, 249 (62,88%) executáveis, 56 casos de teste aleatórios alcançam a cobertura de 58,08%, bastante expressiva, executando 230 elementos e restando 19 elementos executáveis. No entanto são necessários mais 14 (24% mais) casos de teste adicionais para cobrir os elementos restantes. A maioria, (11 elementos dos 19) exigiu um dado de teste em particular. Isso dá uma base de 1,35 elementos cobertos por casos de teste. Antes essa base era de 4,11 elementos por caso de teste. Concluindo, a cobertura, desses elementos difíceis implica num acréscimo razoável do número de casos de teste.

Essa cobertura adicional, é muitas vezes determinante para revelar mais erros. Analisando a eficácia dos dados gerados, percebe-se que os dados "ad-hoc" revelaram cerca de 3% a mais de erros; essa proporção se mantém considerando-se ambos os critérios PU e PU-R. Se for considerada a categoria dos erros revelados, essa diferença aumenta para 3,7% para PU e 5,4% para PU-R quando

Tabela 5.9: Cobertura para cada Função do Programa *comm*

Estrat. Geração	Função	Crit.	Elem. Req.	Elem. Exe.	Elem. N.Exe.	Cobertura	No.Casos Utiliz.
Ad-hoc	compare	PU	5	5	0	100,00	4
		PU-R	10	9	1	90,00	4
	copy	PU	2	2	0	100,00	1
		PU-R	4	3	1	75,00	1
	main	PU	396	249	147	62,88	67
		PU-R	820	448	372	54,63	71
	openfil	PU	6	4	2	66,67	2
		PU-R	9	5	4	55,56	2
	rd	PU	11	11	0	100,00	2
		PU-R	26	25	1	96,15	4
	wr	PU	7	6	1	85,71	5
		PU-R	15	11	4	73,33	5
Aleatória	compare	PU	5	5	0	100,00	2
		PU-R	10	9	1	90,00	2
	copy	PU	2	2	0	100,00	2
		PU-R	4	3	1	75,00	2
	main	PU	396	230	166	51,23	56
		PU-R	820	416	404	50,73	57
	openfil	PU	6	6	0	100,00	2
		PU-R	9	9	0	100,00	2
	rd	PU	11	9	2	80,56	1
		PU-R	26	16	10	61,53	1
	wr	PU	7	6	1	85,71	5
		PU-R	15	11	4	73,33	5

considerados erros de domínio, e é inexistente para erros em estrutura de dados. Isso mostra que os critérios PU e PU-R são fracos em determinar erros em estruturas de dados, independentemente da estratégia de geração utilizada.

A geração aleatória é mais simples e mais barata. Gerar dados "ad-hoc" pode ser mais custoso em termos do número de casos de teste e esforço gasto para gerar um dado específico para cobrir um elemento restante; além do esforço para determinar elementos não executáveis. Mas essa etapa pode levar à revelação de mais erros, erros em geral difíceis de serem determinados, e proporcionar um aprendizado sobre o programa a ser utilizado nas fases de depuração e manutenção e ela não deve ser desprezada. Uma combinação das duas estratégias parece ser o ideal. Primeiramente dados aleatórios, ou funcionais, são gerados, conseguindo-se uma cobertura inicial; para os elementos restantes a estratégia "ad-hoc" é utilizada.

5.2.3 Eficácia: Todos-potenciais-usos X Todos-potenciais-usos-restritos

Como visto na Seção 5.2.1, o critério PU-R requereu 30.10% em média, mais casos de teste que o critério PU. Além disso, o critério PU-R foi mais eficaz. Com os dados "ad-hoc", ele revelou 1,03% a mais de erros. Essa diferença sobe para aproximadamente 1,7% quando considerados apenas erros de domínio. O critério PU-R se mostrou mais eficaz quanto aos erros de domínio por causa dos tipos de restrições escolhidas. Elas foram derivadas a partir da técnica BRO que descreve erros em predicados, ou seja erros de domínio. Abaixo são apresentados alguns exemplos dessas situações nas quais o critério PU-R, utilizando as restrições BRO, se mostrou mais eficaz que o critério PU.

Programa cal

versão incorreta: $if(y > 9999)$

versão correta: $if(y < 1 || y > 9999)$

elem.restrito: $(1,(14,15),\{argc\},(y < 1 \&\& y < 9999))$

Programa crypt

versão incorreta: $if(argc > 1 \&\& argv[1][1] ==' s'){$

versão correta: $if(argc > 1 \&\& argv[1][0] ==' -' \&\& argv[1][1] ==' s'){$

elem.restrito: $(1,(1,3),\{argc\},(argc > 1 \&\& argv[1][0] >' -' \&\& argv[1][1] ==' s'))$

$(1,(1,3),\{argc\},(argc > 1 \&\& argv[1][0] <' -' \&\& argv[1][1] ==' s'))$

No entanto, um problema foi observado no modo como os elementos-restritos foram gerados durante o experimento. Veja o que acontece para uma versão incorreta do programa *spline*.

Programa spline

versão incorreta: $if(i >= 1)$

versão correta: $if(i > 1)$

elem.restrito: nenhum

Não existe nenhum elemento restrito associado ao erro. Isso porque a versão inicial de *Cb-elem* apresentou um erro de implementação. Veja na Figura 5.1 o trecho do programa *spline* correspondente ao predicado acima e o trecho do grafo de fluxo de controle a ele associado. Note que o arco (19,20) será sempre executado ao se executar (20,21) ou (20,22). Os últimos são arcos

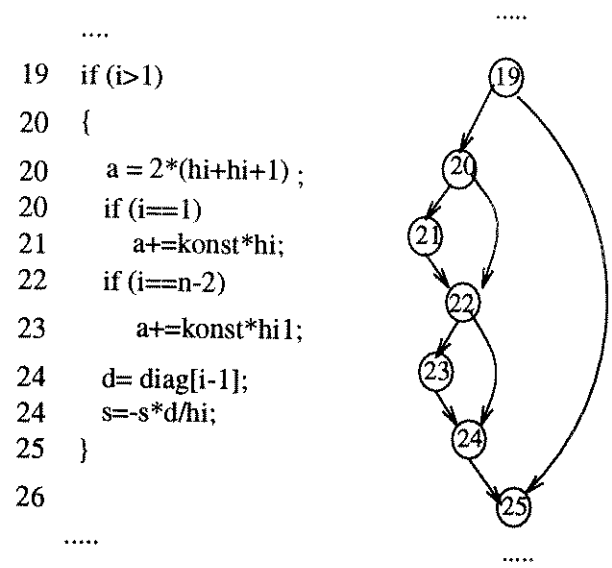


Figura 5.1: Trecho do Programa *spline* - Conceito de Arco Primitivo

primitivos, e, somente associações com relação aos arcos primitivos são requeridas pela POKE-TOOL. As outras são garantidamente cobertas quando as demais o forem. Mas isso não é válido para os critérios restritos, visto que as restrições são diferentes. Portanto nenhuma a associação do tipo $((-,(19,20),-)(i=1))$ que revelaria o erro, foi requerida. Esse problema, observado durante o experimento, aconteceu duas vezes para o programa *spline*, e será corrigido na implementação do módulo *Cb-elemen*.

Os critérios se mostraram pouco eficazes em revelar erros em estruturas de dados (apenas 66,67% deles). Os critérios estruturais não possuem o objetivo de revelar esse tipo de erro e as restrições escolhidas também não os descreviam. Isso mostra a importância dos fatores considerados para a escolha das restrições, a eficácia dos critérios restritos dependerá das restrições escolhidas.

5.2.4 Não Executabilidade

Durante a geração dos dados "ad-hoc", foram identificados os elementos não executáveis requeridos pelos critérios PU e PU-R. A porcentagem de elementos não executáveis obtida foi de 31% e 41%, respectivamente para PU e PU-R. Elas se comparam às porcentagens obtidas em outros experimentos e com outros critérios de teste mais exigentes que o PU [VMJ95b], [Mal91], [Wey93].

Uma associação-restrita é não executável pelo mesmo motivo que a sua associação correspondente o é; mas embora a associação correspondente seja executável, muitas vezes a restrição a tornará não executável. O porque disso, foi objeto de estudo durante o experimento, e deve ser considerado na geração das restrições e dos elementos restritos. Apresenta-se a seguir um resumo das principais causas que geraram associações-restritas não executáveis.

1. Existência de uma intervalo de valores para variáveis e para valores de retorno de funções. Por exemplo, muitos contadores inteiros são inicializados com valor 0 e são sempre incrementados. Restrições que exigem valores negativos para tais contadores não podem ser satisfeitas.
2. Restrições BRO inconsistentes. Por exemplo, a restrição $(c == 9\&\&c > 32)$. Nenhum tratamento desse problema foi realizado durante o experimento.
3. Contadores de laço têm seu valor limitado ao número de interações dos laços. Por exemplo, no laço estabelecido pelo comando $for(i = 0; i < n; i++)$, i nunca será maior que n, pois quando $i = n$, o laço termina e i não é mais incrementado. Para esse predicado, no entanto, as restrições BRO requeridas são: $(i = n, i > n, i < n)$.

Em outros casos, a restrição associada ao elemento era inconsistente com a condição de caminhos livres de definição que cobriam o elemento dado. Os casos enumerados acima devem ser estudados para não serem requeridos elementos restritos não executáveis devido à essas causas. O caso 3 parece ser fácil de ser identificado estaticamente; já nos casos 1 e 2 a identificação dos intervalos de valores das variáveis e valores de retorno de função, poderá ser resultante de uma interpretação da semântica do programa, sendo assim, causas mais difíceis de serem eliminadas.

5.3 Considerações Finais

O experimento descrito no presente capítulo fornece evidências da aplicabilidade dos critérios restritos em programas reais e de relativa complexidade. Mostrou-se que o número de casos de teste necessários não tende a crescer proporcionalmente ao número de elementos requeridos, visto a existência de elementos não executáveis. Vale ressaltar que a determinação da não executabilidade de elementos restritos foi mais fácil de ser realizada, visto que conhecimento necessário sobre o programa já havia sido adquirido durante a aplicação do critério PU.

O critério todos potenciais-usos-restritos (PU-R) quando comparado com o critério todos-potenciais usos (PU) (o correspondente critério estrutural), se mostrou como qualquer outro critério mais exigente. Foram obtidos um maior número de elementos requeridos, e conseqüentemente um maior número de casos de teste necessários e de elementos não executáveis. Um fato importante diz respeito a eficácia dos critérios restritos. Eles foram mais eficazes. Considerando o tipo de erro descrito pelas restrições escolhidas (restrições BRO - erros de domínio) nota-se que o critério PU-R foi mais eficaz que o critério PU. Essa porcentagem é bastante significativa e justifica a aplicação dos critérios restritos PU-R, mais exigentes que PU, em sistemas onde uma alta confiabilidade é requerida.

Uma observação importante diz respeito à dificuldade de revelação dos erros introduzidos nos programas. Para muitos dos programas, a maioria dos casos de teste revela todos os erros

introduzidos. São os casos dos programas:*col*, *comm*, *look*, *tr*. Por exemplo, para o programa *tr*, 26,67% dos casos de teste efetivos (8 em 30) para o critério PU, gerados de maneira "ad-hoc", revelam todos os 19 erros introduzidos no programa. Os erros de programas como *crypt*, *uniq*, *spline* se mostraram mais difíceis de serem revelados. Em tais programas a diferença de eficácia entre critérios PU e PU-R, e entre dados aleatórios e "ad-hoc", foi maior.

Um outro resultado obtido com esse experimento foi verificar, na prática, a influência das restrições utilizadas na eficácia dos critérios restritos. As restrições escolhidas descrevem erros de domínio. Para esses erros, o critério PU-R se mostrou eficaz. Isso não aconteceu para os erros em estrutura de dados. O critério PU não se mostrou bastante forte para revelar esse tipo de erro; apenas 66,67% desse erros puderam ser revelados. Para o critério PU-R, nenhuma restrição descrevendo esse tipo de erro foi associada às associações requeridas.

O estudo dos elementos restritos não executáveis foi importante. As causas apresentadas na Seção 5.2.4 serão consideradas para gerar um número menor de restrições não executáveis, quando o módulo *Cb-kernel*, descrito no capítulo anterior, for implementado. Também os módulos *Pokeheuris* e *Pokepadrao* poderão ser futuramente adaptados para tratamento de elementos restritos não executáveis.

A estratégia aleatória apesar de não garantir a cobertura dos critérios, mostrou ser prática e conseguiu porcentagens de cobertura e eficácia bastante significativas. No entanto, executar os elementos restantes e gerar dados "ad-hoc" leva a uma maior cobertura e à satisfação dos critérios e, como foi comprovado, a uma maior eficácia em termos da revelação de erros. O ideal é combinar essas estratégias e explorar os seus aspectos positivos.

Capítulo 6

Uma Estratégia Incremental para Geração de Dados de Teste

Um critério de teste estrutural pode ser utilizado como um critério de adequação de um dado conjunto de testes, e/ou como um critério para geração de dados de teste. Ele poderá ser utilizado para avaliar a adequação de um conjunto inicial. Se for desejado aprimorar esse conjunto, utiliza-se o critério para geração.

No Capítulo 2 foram abordadas as principais limitações e estratégias existentes para realizar a atividade de geração de dados de teste para satisfazer os diferentes critérios. Vale ressaltar a importância dessa atividade, visto que a eficácia do critério, ou seja, revelar ou não um erro, pode ser dependente da estratégia de geração utilizada.

Os Critérios Restritos foram definidos com o objetivo de aumentar a eficácia das atividades de teste. Extensões propostas à ferramenta POKE-TOOL apoiam a utilização dos Critérios Restritos e facilitam a atividade de geração de dados de teste.

Este capítulo é resultado de estudos realizados e da aplicação das idéias propostas nesta dissertação. Ele sintetiza toda a experiência observada durante a condução de diversos experimentos empíricos [Ver92, VMJ93a, VMJ95b, VMJ96a], e dos experimentos descritos nos Capítulos 3 e 5. São fornecidas algumas diretrizes a serem consideradas para realizar a tarefa de geração de dados de teste, e duas estratégias são propostas. A primeira, denominada SGER (Estratégia para **G**eração de Dados de Teste Sensíveis a **Er**ros), fornece um roteiro para a aplicação das diretrizes propostas e para a utilização dos Critérios Restritos. A segunda, denominda SINGER (Estratégia **In**cremental para **G**eração de Dados de Teste Sensíveis a **Er**ros), está baseada na hierarquia entre critérios de teste, e tem o objetivo de garantir que a relação de inclusão seja sempre preservada, mesmo quando considerado o fator eficácia.

6.1 Diretrizes para Gerar Dados de Teste

A seguir são feitas algumas considerações sobre a atividade de geração de dados de teste para satisfazer critérios de teste estrutural. Elas estão organizadas em forma de diretrizes básicas, e muitas delas foram utilizadas para o estabelecimento da nova arquitetura da ferramenta POKE-TOOL, descrita no Capítulo 4. Elas visam diminuir o esforço gasto na geração de dados de teste, e facilitar a sua automatização, com o objetivo de:

- reduzir o número de dados de teste selecionados para satisfazer um critério;
- reduzir os efeitos causados por caminhos não executáveis, fornecendo auxílio para determiná-los, e evitar que tempo e esforço sejam gastos na tentativa de gerar um dado de teste para cobrí-los;
- revelar um número maior de erros.

6.1.1 O número de dados de teste necessários

Essa questão diz respeito ao número de dados de teste necessários para satisfazer um critério e à ordem de execução dos elementos requeridos.

- *Diretriz 1*: Se a cobertura do elemento E_1 garante a cobertura do elemento E_2 execute primeiro E_1.

Muitas vezes a cobertura de um elemento garante a cobertura de vários outros. Portanto, ele deverá ser executado primeiro. Isso pode levar a um número menor de caminhos completos a serem executados, e a um número menor de casos de teste.

6.1.2 Não executabilidade

Um dos maiores problemas encontrados na tarefa de geração de dados de teste é a não executabilidade de caminhos. Muitas vezes, entre os caminhos selecionados para cobrir os elementos requeridos por um critério, existem muitos caminhos não executáveis. Tempo e esforço são gastos na tentativa de gerar dados de teste para esses caminhos. É desejável selecionar um conjunto contendo apenas caminhos executáveis, e poder identificar a executabilidade dos elementos requeridos.

- *Diretriz 2*: Utilize ferramentas que ofereçam tratamento e eliminação de não executabilidade.

A nova versão scripts-UNIX da ferramenta POKE-TOOL oferece tais facilidades através dos módulos *Pokepadrao*, *Pokeheuris* e *Pokefatos*.

• *Diretriz 3*: Entre dois caminhos candidatos a cobrir um elemento requerido, escolha o de menor número de predicados porque ele tem menor probabilidade de ser não executável.

Essa diretriz está baseada nos trabalhos de Malevris et al [MYV90] e Vergilio et al [Ver92, VMJ93a] referenciados no Capítulo 2.

6.1.3 Revelação de um número maior de erros

Essa questão diz respeito à eficácia dos dados selecionados para executar os caminhos requeridos. A eficácia dos dados pode ser dependente da estratégia escolhida. Além da condição do caminho a cobrir o elemento requerido, os dados devem satisfazer outras condições que descrevam um erro específico, ou uma classe de erros.

• *Diretriz 4*: Utilize critérios restritos para que os dados de teste gerados tenham uma maior probabilidade de revelar erros.

6.2 Utilizando as Diretrizes Propostas - Estratégia SGER

Nessa seção é apresentada a estratégia SGER (Estratégia para Geração de Dados de Teste Sensíveis a **Erros**). Ela consiste de uma seqüência de passos a ser seguida para bem utilizar as diretrizes propostas. As diretrizes apresentadas na seção anterior são independentes do critério de teste utilizado, elas podem ser aplicadas com qualquer critério de teste estrutural. Para ilustração da estratégia SGER, utiliza-se o programa *append*, extraído de [KP81] e a ferramenta de testes POKE-TOOL. O programa *append* e o seu *grafo-def* estão na Figura 6.1. A seguir são apresentadas as principais questões que estão relacionadas com a cobertura das associações requeridas pelo critério todos-potenciais-usos.

Como gerar dados de teste para cobrir uma associação?

1. *Dada uma lista de associações exigidas pelo critério, qual associação selecionar?*

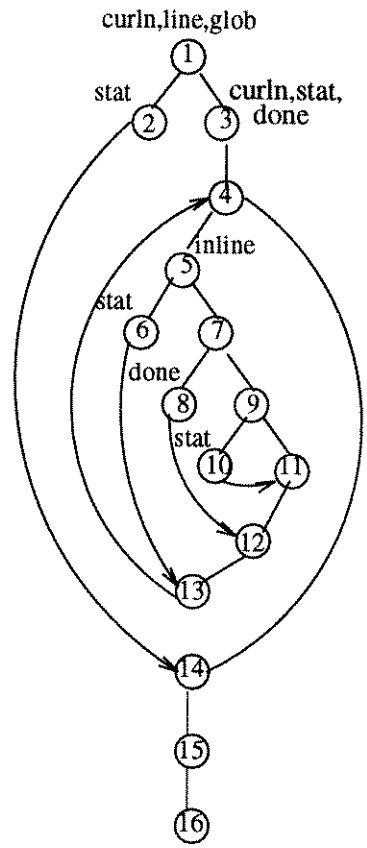
• Primeiramente utilizar *diretriz 2*. Utilizar rotinas de tratamento de não executabilidade da ferramenta POKE-TOOL. Por exemplo, na Tabela 6.1, encontram-se as associações requeridas pelos critérios Potenciais Usos, e geradas pela POKE-TOOL para o exemplo. O módulo *Pokeheuris* pode nesse caso determinar automaticamente todas as associações não executáveis (34% das associações requeridas), que não farão parte da lista que deve ser considerada, e estão assinaladas com um (*) na Tabela 6.1.

• Para selecionar as associações considere agora a *diretriz 1*. Entre dois conjuntos c_1: $(i,(j,k),V_1)$ e c_2: $(i,(j,k),V_2)$, (analogamente para (i,j,V_1) e (i,j,V_2)), sendo V_1 e V_2 conjuntos de variáveis tal que $V_2 \subset V_1$, selecionar o conjunto $c1$, pois se c_1 for coberto, c_2 também o será, uma

```
typedef enum {ENDDATA,ERR,OK} stcode;
int curln;

stcode append(int line, int glob)
 {  char inline[MAXSTR];
    stcode stat, int done;
1   if (glob)
2     stat=ERR;
3   else
3    { curln=line;stat=OK;done=0;
4     while((!done)&&(stat==OK))
5     {
5       if(!getline(inline,stdin,MAXSTR))
6          stat=ENDDATA  ;
7       else
        {
7         if((inline[0]==PERIOD)&&
            (inline[1]==NEWLINE))
8           done=1;
9         else
        {   if(puttxt(inline)==ERR)
10            stat=ERR;
11      }
12    }
13   }
14  }
15  return stat;
16 }
```

Program Correto



Grafo de Fluxo de Controle

Figura 6.1: Programa Append - Uso das Diretrizes de Geração de Dados de Teste

Tabela 6.1: Associações Requeridas - Critério Todos-potenciais-usos

1)<1,(4,14),{line,glob}>	18)<5,(9,11),{inline}>	35)<8,(6,13),{done}> *
2)<1,(9,11),{line,glob}>	19)<5,(9,10),{inline}>	36)<8,(5,6),{done}>*
3)<1,(9,10),{line,glob}>	20)<5,(7,8),{inline}>	37)<9,(9,11),{inline}>
4)<1,(7,8),{line,glob}>	21)<5,(4,14),{inline}>	38)<9,(4,14),{inline}>
5)<1,(13,4),{line,glob}>	22)<5,(4,5),{inline}>	39)<9,(7,9),{inline}>
6)<1,(5,6),{line,glob}>	23)<5,(5,6),{inline}>	40)<9,(8,12),{inline}>
7)<1,(1,3),{curln,line,glob }>	24)<6,(4,14),{stat}>	41)<9,(7,8),{inline}>
8)<1,(1,2),{curln,line,glob}>	25)<6,(9,11),{stat}>*	42)<9,(6,13),{inline}>
9)<2,(,),{stat}>	26)<6,(9,10),{stat}>*	43)<9,(5,6),{inline}>
10)<3,(4,14),{curln,stat,done}>*	27)<6,(12,13),{stat}>*	44)<9,(9,10),{inline}>
11)<3,(9,11),{curln,stat,done}>	28)<6,(7,8),{stat}>*	45)<10,(4,14),{stat}>
12)<3,(9,10),{curln,stat,done}>	29)<6,(5,6),{stat}>*	46)<10,(9,11),{stat}>*
13)<3,(7,8),{curln,stat,done}>	30)<8,(4,14),{done}>	47)<10,(9,10),{stat}>*
14)<3,(13,4),{curln,stat,done}>	31)<8,(9,11),{done}>*	48)<10,(8,12),{stat}>*
15)<3,(13,4),{curln,stat}>	32)<8,(11,12),{done}> *	49)<10,(7,8),{stat}>*
16)<3,(13,4),{curln,done}>	33)<8,(9,10),{done}>*	50)<10,(5,6),{stat}>*
17)<3,(5,6),{curln,stat,done}>	34)<8,(7,8),{done}>*	

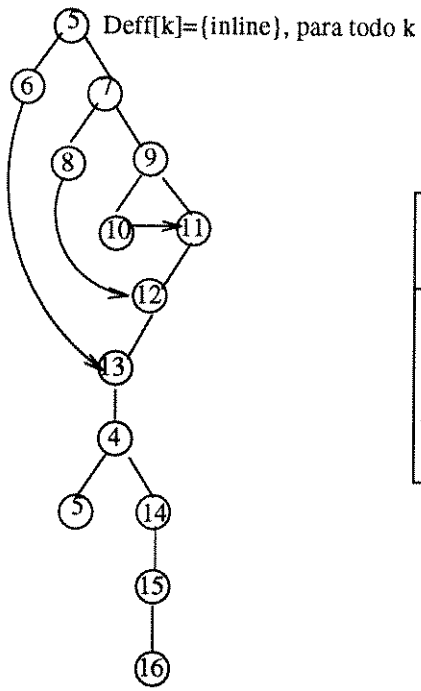
vez que todo caminho livre de definição com respeito às variáveis de V_1, é também livre de definição com respeito às variáveis de V_2, mas o contrário nem sempre acontece. Por exemplo, na geração dos dados de teste para cobrir as associações 14, 15, 16 (Tabela 6.1), a associação número 14 deverá ser considerada primeiro, pois todo caminho livre de definição c.r. às variáveis {curln,stat,done} é livre de definição c.r.a {curln,stat} e c.r.a {curln,done} e cobrirá também as associações 15 e 16. Se a associação 14 não fosse selecionada primeiro, poderia ser necessária, no pior caso, a execução de três caminhos.

2.*Qual caminho cobre uma associação?*

• São caminhos completos que contêm sub-caminhos livres de definição com respeito às variáveis da associação de i para (j,k). O módulo *Pokepaths* da POKE-TOOL determinará esses caminhos

3.*Qual caminho completo selecionar?*

• Novamente, o módulo *Pokepaths*, permite que uma estratégia de seleção dos caminhos seja escolhida. Utilizar *diretriz 3* e escolher menor número de predicados. Por exemplo, para o programa *append*, tomando-se a associação $(5,(4,14),\{inline\})$, tem-se, utilizando-se o grafo(5) gerado pela POKE-TOOL (Figura 6.2), os seguintes potenciais-du-caminhos: 5 6 13 4 14, 5 7 9 10 11 12 13 4 14, 5 7 9 11 12 13 4 14, e 5 7 8 12 13 4 14, livres de definição c.r.a *inline*. Para todos os caminhos completos gerados o laço do nó 4 deve ser considerado, portanto tomar-se-á



Potencias-du-caminhos para [5,(4,14),inline]	número de predicados
5 6 13 4 14	3
5 7 8 12 13 4 14	5
5 7 9 10 11 12 13 4 14	6
5 7 9 11 12 13 4 14	6

Figura 6.2: Grafo-i Gerado para o Nó 5 - Uso das Diretrizes de Geração de Dados de Teste

como caminho com menor número de predicados através do laço do nó 4, aquele que não percorre o laço. Os caminhos então diferem por causa das diferentes maneiras de ir do nó 5 ao arco (4,14), dadas pelo grafo(5) (Figura 6.2). Assim, o caminho completo escolhido é 1 3 4 5 6 13 4 14 15 16. Note que pode-se evitar que o caminho não executável 1 3 4 5 7 9 11 12 13 4 14 15 16, um dos que possui maior número de predicados, seja escolhido.

4.*Como gerar o dado de teste para executar o caminho selecionado?*

Utilize diretriz 4. Considere que o *if* do nó 7 do programa *append* possui um erro no operador relacional: $inline[0] <= PERIOD$. Na Tabela 6.2 estão alguns casos de teste para o programa *append*. Se o conjunto de casos de teste {1,2,3,4} for selecionado, o erro não vai ser revelado. No entanto o erro será descoberto se o conjunto {1,2,3,5} for selecionado. Isso porque o caso de teste 5 é o único que satisfaz a restrição ($inline[0] < PERIOD$). Essa restrição é uma restrição BRO, e pode ser facilmente considerada para gerar os critérios restritos, como foi considerado no Capítulo 5.

Tabela 6.2: Casos de Teste para o Programa *append*

número	valores para glob e line inline[]	saída esperada	saída obtida	caminho percorrido
1	0 0 digitando <ret> qualquer coisa <ret>	0	0	1 3 4 5 7 9 11 12 13 4 5 7 9 11 12 13 4 5 6 13 4 14 15 16
2	0 15 qualquer coisa <ret>	1	1	1 3 4 5 7 9 10 11 12 13 4 14 15 16
3	1 0	1	1	1 2 15 16
4	0 0 a<ret> .<ret>	2	2	1 3 4 5 7 9 11 12 13 4 5 7 8 12 13 4 14 15 16
5	0 0 a<ret> +<ret>	0	2	1 3 4 5 7 9 11 12 13 4 5 7 8 12 13 4 14 15 16

6.3 Uma Estratégia Baseada na Hierarquia entre os Critérios de Teste

Nessa seção é proposta uma estratégia, denominada SINGER (Estratégia **I**ncremental para **G**eração de Dados de Teste Sensíveis a **Er**ros), que considera a dificuldade de satisfação dos critérios e a hierarquia entre eles, dada pela relação de inclusão. Ao se satisfazer um critério exigente tal como todos-potenciais-usos, um critério menos exigente tal como todos-ramos, também é satisfeito. A estratégia consiste em começar o teste, aplicando-se o critério menos exigente e utilizando sempre que possível as diretrizes apresentadas. Dado o conjunto de critérios $(C_1, C_2, .., C_i, .., C_n)$ tal que $C_i \Rightarrow C_{i-1}$ (para $2 \leq i \leq n$), parte-se do critério C_1 (o mais abaixo na hierarquia), gera-se um conjunto T_1 de casos de teste, C_1-adequado, que revela um conjunto de defeitos F_1. Para o critério C_2 gera-se um conjunto T_2, C_2-adequado, tal que $T_1 \subset T_2$, que revelará um conjunto de defeitos F_2, tal que $F_1 \subset F_2$. Essa estratégia permite que a hierarquia seja sempre preservada. Se $C_i \Rightarrow C_j$, e $T_j \subset T_i$, preserva-se a hierarquia pois garante-se que $F_j \subset Fi$, ou seja, que todos os defeitos revelados por C_j sejam revelados também por C_i.

Para aplicação dos critérios, pode-se considerar uma hierarquia empírica, que reflete a dificuldade de satisfação dos critérios. Por exemplo, a probabilidade de satisfazer o critério todos-potenciais-usos é maior dado o fato do critério todos-ramos ter sido satisfeito, do que dado o fato do critério todos-nós ter sido satisfeito. Isso porque é necessário cobrir todos os ramos para se satisfazer o critério todos-potenciais-usos. Os critérios restritos são mais difíceis de serem satisfeitos que os critérios estruturais correspondentes. Os Critérios Restritos e o Critério Análise de Mutantes são

incomparáveis. A dificuldade de satisfação desses critérios dependerá respectivamente das restrições e/ou níveis escolhidos e dos operadores de mutação selecionados.

Quanto ao fator eficácia, também nota-se em experimentos empíricos que existe um limite de saturação para cada critério [MAR92, Cre97]. Numa primeira etapa, durante a aplicação de um critério, uma alta porcentagem de erros é revelada e esse número tende a cair com o aumento da cobertura, e estacionar, chegando a um ponto de saturação, onde embora a cobertura aumente, nenhum erro é revelado. Nesse momento, a aplicação de um critério mais exigente pode levar a um crescimento da taxa de erros revelados. Sugere-se que os Critérios Restritos sejam aplicados após os correspondentes critérios estruturais. Conforme os resultados apresentados no Capítulo 5, isso poderia levar a um aumento de cerca de 1.7% no número de erros revelados, o que é considerado bastante significativo [Cre97].

6.4 Considerações Finais

Esse capítulo sintetiza toda a experiência obtida durante o desenvolvimento do trabalho descrito nessa dissertação. Diretrizes para a geração de dados de teste são propostas com o objetivo de reduzir os efeitos causados por caminhos não executáveis, reduzir o número de casos de teste, e facilitar a etapa de geração de dados de teste. Duas estratégias foram propostas: a estratégia SGER, que fornece uma seqüência de passos a serem seguidos; e a estratégia SINGER, que consiste na aplicação dos critérios segundo a hierarquia dada pela relação de inclusão.

Quando um critério é escolhido, e aplicado independentemente, a utilização da estratégia SGER permite que dados com uma maior probabilidade de revelar erros sejam gerados, e como visto no Capítulo 3, a utilização dos Critérios Restritos pode contribuir, em muitos casos, para preservar a hierarquia entre os critérios. Aplicando-se a estratégia incremental SINGER, garante-se que a hierarquia entre os critérios seja preservada, quando considerado o fator eficácia. A obrigatoriedade de se aplicar todos os critérios não se constitui uma desvantagem; dependendo da criticalidade da aplicação, a utilização de todos os critérios se faz necessária, i.e, quando uma alta confiabilidade for requerida, o custo é justificado, pois a aplicação da estratégia contribui para revelar um número maior de erros, e para um aumento da qualidade dos testes gerados.

O módulo *Pokepaths* (descrito no Capítulo 4) está sendo implementado e foi planejado um experimento para validar a utilidade da diretriz 3 [Per97]. Pretende-se verificar, para os programas utilizados no experimento descrito no Capítulo 5, o quanto essa diretriz contribui para evitar caminhos não executáveis, e também estudar o quanto as estratégias menor número predicados e/ou menor número de casos de teste influenciam (a ponto de diminuir) a eficácia dos critérios.

Capítulo 7

Conclusões e Trabalhos Futuros

O trabalho desta dissertação abordou os temas critérios estruturais e estratégias de geração de dados de teste, uma tarefa que consome muito esforço e que possui muitas limitações para sua completa automatização.

Foram realizados estudos empíricos e teóricos comparando as principais técnicas de geração de dados de teste sensíveis a erros. Durante esses estudos pôde-se comprovar o aspecto complementar dessas técnicas, as principais limitações e dificuldades de aplicação, discutidos no Capítulo 2. Os Critérios Restritos, introduzidos no Capítulo 3, são resultado da experiência obtida com esses estudos. Eles permitem combinar diferentes princípios e conceitos de teste com os critérios estruturais, permitindo explorar as características vantajosas das diferentes técnicas, dando assim mais poder ao teste estrutural. Além disso, como mostrado experimentalmente, a combinação das restrições de necessidade que descrevem esses erros, com as condições dos caminhos para cobrir os elementos requeridos, muitas vezes satisfaz a condição de suficiência para que o erro seja revelado, sendo esta uma das vantagens em se aplicar Critérios Restritos, em comparação com outra técnica de geração dissociada de um critério estrutural.

Do projeto de aplicação piloto conduzido com os Critérios Restritos, e considerando os três primeiros níveis para a escolha das restrições, foram obtidos resultados promissores: o número de casos de teste exigidos pelos critérios restritos não tende a crescer tanto quanto o número de elementos requeridos, e em todos os casos foi menor ou igual ao número requerido pelas estratégias DT e CBT; todos os erros considerados puderam ser revelados pelo critério todos potenciais-usos restritos utilizando-se os três primeiros níveis para a escolha das restrições.

Em face desses resultados foram propostas extensões à ferramenta POKE-TOOL propondo-se novos módulos. Os módulos propostos fornecem apoio à utilização dos critérios Potenciais-Usos Restritos e facilitam a etapa de geração, fornecendo caminhos completos para cobrir um elemento (ou um dado conjunto de elementos). O módulo *Cb-aval* foi implementado e o modelo de instrumentação para a inserção das restrições pôde ser validado.

Utilizando-se o módulo de avaliação dos Critérios Restritos, realizou-se o experimento descrito no Capítulo 5, com programas bem mais complexos que os utilizados no Capítulo 3, e com apenas um tipo de restrição, restrições BRO. Resultados similares foram obtidos, e fornecem evidências da aplicabilidade dos critérios baseados em restrições, em programas reais. Para o critério todos potenciais-usos restritos (PU-R) foi obtida uma menor cobertura, foi requerido um maior número de elementos, e conseqüentemente, um maior número de casos de teste foi necessário, e um maior número de elementos não executáveis foi encontrado, quando comparado com o critério todos-potenciais usos (PU), o correspondente critério estrutural. Esses resultados são explicados pela relação de inclusão, um critério restrito inclui o seu critério estrutural correspondente. Um fato importante diz respeito à eficácia dos critérios restritos. Considerando o tipo de erro descrito pelas restrições escolhidas (restrições BRO - erros de domínio) nota-se que o critério PU-R revelou 1.7 mais erros que o PU. Essa porcentagem é bastante significativa e justifica a aplicação dos critérios restritos em sistemas onde uma alta confiabilidade é requerida.

Vale ressaltar a influência do tipo de restrição associada ao critério estrutural no aumento de sua eficácia. Devem ser associados aos elementos requeridos, restrições que descrevem erros complementares aos revelados pelas técnicas estruturais, por exemplo, que descrevem erros em estrutura de dados. A estruturação das restrições em níveis, a serem considerados durante a escolha das mesmas, é uma boa sugestão para se saber qual a informação necessária, quais técnicas, e conseqüentemente, quais tipos de erros são descritos em cada nível. A escolha das restrições é uma das tarefas mais importantes. Além dos níveis propostos para a escolha da restrição, sugere-se que um tipo de critério-restrito seletivo seja utilizado, considerando vários fatores: o tipo de programa em teste e tipos de erros aos quais ele está sujeito, a confiabilidade requerida, o tempo disponível, custos, etc. Isso pode leva a uma redução dos custos, diminuindo o número de elementos requeridos, e o número de casos de teste.

Ainda observou-se com o experimento do Capítulo 5, que a estratégia aleatória é bastante prática, mas como foi observado na Seção 5.2.2, os elementos mais difíceis de serem cobertos não são, em geral, cobertos por esse tipo de dados. Esses elementos são cobertos por um dado específico, e podem significar partes do programa pouco executadas e sujeitas a erros não facilmente revelados; por isso, o ideal é uma estratégia de geração que combine ambas as estratégias apresentadas e que possa ser utilizada para ambos os critérios PU e PU-R. Um conjunto inicial de dados aleatórios é utilizado, e uma cobertura inicial é obtida; dados são gerados "ad-hoc" para os elementos restantes. Nesta etapa, um aprendizado sobre o programa é realizado e esse conhecimento poderá depois ser utilizado durante o teste de regressão, nas fases de depuração e manutenção.

7.1 Contribuições

As principais contribuições desse trabalho são enumeradas a seguir, de acordo com os diferentes tópicos abordados nos capítulos dessa dissertação.

1. *Estudos comparativos das diferentes técnicas de geração de dados de teste sensíveis a erros*: foram realizados estudos teóricos e práticos. Durante o experimento observou-se um maior poder em termos de eficácia das técnicas Domain Testing e Constraint Based Testing, e uma maior praticidade da técnica BRO/BOR, em termos de facilidade de aplicação e número de casos de teste requeridos. Salienta-se as limitações dessas estratégias na presença de correção coincidente e caminhos ausentes. Outro fato observado nesses estudos, foi uma maior dificuldade na determinação de condições de suficiência para erros de computação.

2. *Definição de Critérios Baseados em Restrições*: a aplicação dos Critérios Restritos introduzidos nessa dissertação: a) leva a um aumento da eficácia dos critérios estruturais correspondentes, possibilitando combinar a técnica estrutural com diferentes princípios de técnicas complementares; b) contribui para preservar a hierarquia entre critérios, quando considerado o fator eficácia e um conjunto comum de restrições; c) oferece medidas de cobertura para quantificar a atividade de teste, e avaliar a adequação de um conjunto de casos de teste.

3. *Extensões propostas à POKE-TOOL*: possibilitam a implementação e aplicação dos Critérios Restritos, além de facilitar a determinação de elementos não executáveis, e estabelecer conjunto de caminhos para satisfação dos critérios estruturais. Esses módulos facilitarão a etapa de geração automática de dados de teste, pois dados de teste para satisfação dos critérios estruturais poderão ser derivados automaticamente, resolvendo-se o conjunto de restrições formado pelas condições dos caminhos e pelas restrições que descrevem os erros, aos quais o programa está sujeito.

4. *Experimento de aplicação dos Critérios Baseados em Restrições*: tanto o experimento piloto, quanto o experimento descrito no Capítulo 5, comprovam a aplicabilidade dos Critérios Restritos. Notou-se não ser muito difícil a identificação da não executabilidade dos elementos restritos, uma vez determinada a não executabilidade dos correspondentes elementos estruturais. O critério todos-potenciais-usos-restritos também se mostrou mais eficaz que o correspondente critério estrutural, revelando uma porcentagem maior de erros, principalmente os erros descritos pelas restrições utilizadas.

5. *Proposição de estratégias para geração de dados de teste*: as estratégias apresentadas no Capítulo 6 podem ou não ser utilizadas conjuntamente, mas sem dúvida estabelecem um bom roteiro para a tarefa de aplicação dos critérios de teste estrutural existentes, incluindo os Critérios Restritos. Visam a aumentar a eficácia e, conseqüentemente, a qualidade dos dados de teste gerados, além de reduzir os custos da atividade de teste. Permitem:

- A redução de esforço de geração de dados de teste, minimizando o número de caminhos para cobrir os elementos requeridos por um dado critério de teste.
- A redução dos efeitos causados por caminhos não executáveis, pois pode-se evitar que esforço seja gasto com alguns elementos não executáveis e que sejam selecionados caminhos com maior probabilidade de serem executáveis.
- Que os dados de teste, gerados para cobrir os elementos requeridos, possuam alta probabilidade de revelar erros, visto que satisfazem suas restrições de necessidade, e têm alta probabilidade de, juntamente com a condição do caminho que cobre o elemento requerido, também satisfazer suas condições de suficiência.
- Se a estratégia incremental for utilizada, garante-se a preservação da hierarquia entre os critérios, quanto ao fator eficácia.

7.2 Trabalhos Futuros

Podem ser visualizados muitos desdobramentos desse trabalho de pesquisa, muitos deles em andamento.

Sobre a escolha das restrições, pretende-se estudar mais detalhadamente que tipos de restrições são mais adequadas, dado um tipo de programa. Esse estudo poderá contribuir para o estabelecimento de Critérios Restritos Seletivos, onde são estabelecidos diferentes grupos de restrições, além dos agrupamentos em níveis sugeridos. Isso poderia levar à redução de custos e facilitar a aplicação dos Critérios Restritos.

Delamaro [Del97] estabelece operadores de mutação para o teste de integração. Restrições de necessidade podem ser derivadas, a partir desses operadores, para que os Critérios Baseados em Restrições possam ser utilizados durante o teste de integração.

Já estão sendo incorporados à versão "scripts-UNIX" da POKE-TOOL, os módulos de tratamento de não executabilidade *Pokeheuris*, *Pokefatos*, *Pokepadrao* [BJVM97]. Também está sendo implementado o módulo *Pokepaths* [Per97]. O próximo passo é implementar o módulo *Cb-kernel* para derivar automaticamente os elementos restritos requeridos. Mais ainda, no futuro pretende-se acoplar à ferramenta POKE-TOOL, outras ferramentas para execução simbólica e/ou dinâmica para, juntamente com as restrições estabelecidas pelos Critérios Baseados em Restrições, poder realizar a geração automática dos dados de teste.

Pretende-se realizar dois outros experimentos. O primeiro experimento está sendo realizado com os programas utilizados no Capítulo 5 [Vin97]. Os dados aleatórios e "ad-hoc", utilizados para satisfazer os critérios todos potenciais-usos e todos potenciais-usos-restritos, serão utilizados com a ferramenta de teste Proteum, os mutantes equivalentes serão determinados e poder-se-á verificar

o "strength" desses critérios em relação ao critério Análise de Mutantes, comparando os resultados PU X PU-R e geração aleatória X geração "ad-hoc".

O outro experimento tem o objetivo de validar as diretrizes e estratégia incremental proposta no Capítulo 6. Ele já foi programado, e com auxílio do módulo *Pokepaths*, tem o objetivo de avaliar a Diretriz 3, para verificar o quanto ela é efetiva para evitar caminhos não executáveis. Além disso, pretende-se estudar a relação entre a eficácia e as estratégias menor número de predicados, maior ou menor complexidade das estruturas de controle envolvidas no caminho [PVJM97] [Per97].

Referências Bibliográficas

[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 simbolic execution. *ACM SigPlan Notices*, Vol. 10(6):234–245, June 1975.

[BJVM97] P.M. Bueno, M. Jino, S.R Vergilio, and J.C. Maldonado. Uma proposta de extensão da ferramenta poke-tool para apoiar o tratamento de não executabilidade. In *Workshop do Projeto Validação e Teste de Sistemas de Operação*, pages 213–222. Águas de Lindóia-SP, January 1997. (in Portuguese).

[BM83] D.L. Bird and C.U. Munoz. Automatic generation of random self-cheking test cases. *IBM System Journal*, Vol. 22(3):229–245, 1983.

[BP84] V.R. Basili and B.T. Pericone. Software erros 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.

[Cha91] M.L. Chaim. *POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados.* Master Thesis, DCA/FEEC/Unicamp, Campinas - SP, Brazil, April 1991. (in Portuguese).

[CHR82] L.A. Clarke, J. Hassell, and D.J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, Vol. SE-8(4):380–390, July 1982.

[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.

[CR83] L.A Clarke and D.J. Richardson. The application of error-sensitive testing strategies to debugging. In *Symposium of High-Level Debugging*, pages 45–52. ACM SIGSOFT/SIGPLAN, May 1983.

[Cra89] W.M. Craft. *Detecting Equivalents Mutants Using Compiler Optimization.* Master Thesis, Department of Computer Science, Clemson University, Clemson-SC, 1989.

[Cre97] A.N. Crespo. *Modelos de Confiabilidade Baseados em Cobertura de Critérios Estruturais.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, 1997. (in elaboration, in Portuguese).

[Dea96] M. E. Delamaro and et al. Integration testing using interface mutation. In *VII International Symposium of Software Reliability Engineering (ISSRE)*, pages 112–121. IEEE Computer Society Press, New York, NY, November 1996.

[Del93] M.E. Delamaro. *Proteum: Um ambiente de teste baseado na Análise de Mutantes.* Master Thesis, ICMSC-USP-São Carlos, São Carlos - SP, Brazil, March 1993. (in Portuguese).

[Del97] M.E. Delamaro. *Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração.* Doctorate Dissertation, ICMSC-SC/USP, São Carlos - SP, 1997. (in Portuguese).

[DMGK88] R.A. De Millo, D.C. Gwind, and K.N. King. An extented overview of the mothra software testing environment. In *Proc. of the Second Workshop on Software Testing, Verification and Analysis*, pages 142–151. Computer Science Press, Banff - Canada, July 19-21 1988.

[DMLS78] R.A. De Millo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, Vol. C-11:34–41, April 1978.

[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 TEX.* Technical Report, CS/Purdue University, September 1995.

[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.

[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.

[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.

[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.

[FW93a] F.G. Frankl and E. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, Vol. SE-19(10):962–975, October 1993.

[FW93b] P.G. Frankl and E.J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, Vol. SE-19(3):202–213, March 1993.

[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.

[GW86] M.R. Girgis and M.R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proc. of the Workshop on Software Testing*, pages 64–71. Computer Science Press, Banff - Canada, July 1986.

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

[HL90] J.R. Horgan and S. London. *ATAC- Automatic Test Coverage Analysis for C Programs*, June 1990.

[How75] W.E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Software Engineering*, Vol. SE-24(5):554–559, May 1975.

[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 19-21 1988.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* New York, 1990.

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

[KP81] B.W. Kernighan and P.J. Plauger. *Software Tools in Pascal.* Addison-Wesley Publishing Company Reading, Massachusetts - USA, 1981.

[LK83] J.W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, Vol. SE-9(3):347–354, May 1983.

[Mal91] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, July 1991. (in Portuguese).

[MAR92] Chen M., Mathur A.P., and V.J. Rego. Effect of testing techniques on software reliability estimates obtained using time domain models. In *Proc. of the 10th Annual Software Reliability Symposium*, pages 116–123. IEEE Reliability Society, Denver - Colorado, June 1992.

[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 testes baseada nos critérios potenciais usos. In *II Simpósio Brasileiro de Engenharia de Software*, pages 24–35. Sociedade Brasileira de Computação - SBC, Canela - RS, Brazil, October 1988. (in Portuguese).

[Mor90] L.J. Morrel. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, Vol. SE-16(8):844–857, August 1990.

[MW93] A.P Mathur and W.E. Wong. Evaluation of the cost of alternate mutation strategies. In *VII Simpósio Brasileiro de Engenharia de Software*, pages 320–375. Rio de Janeiro-Brazil, October 1993.

[Mye79] G. J. Myers. *The Art of Software Testing.* Wiley, 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.

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

[Oea96] A.J. Offut and et al. An experimental evaluation of sufficient mutant operators. *ACM-Transactions on Software Engineering Methodology*, Vol. 5(2):99–118, April 1996.

[Oea97] A.J. Offut and et al. An experimental evaluation of sufficient mutant operators. *Software Practice and Experience*, 1997. (to appear).

[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.

[Per97] L.M. Peres. *Estudo de Estratégias de Seleção de Caminhos para Satisfação de Critérios de Teste Estrutural.* Master Thesis, DCA/FEEC/Unicamp, Campinas - SP, 1997. (in elaboration, in Portuguese).

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

[PVJM97] L.M. Peres, S.R Vergilio, M. Jino, and J.C. Maldonado. Aspectos de seleção de caminhos para cobertura de critérios estruturais de teste. In *Workshop do Projeto Validação e Teste de Sistemas de Operação*, pages 113–122. Águas de Lindóia-SP, January 1997. (in Portuguese).

[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.

[RT88] D.J. Richard and M.C. Thompson. The relay model for error detection and its application. In *Proc. of 2nd Workshop on Software, Testing, Verification and Analysis*, pages 223–230. Los Alamitos, IEEE Computer Society, Banff - Canada, July 1988.

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

[Sil95] J.B. Silva. *Proteste+: Ambiente de Validação Automática de Qualidade de Software através de Técnicas de Teste e de Métricas de Complexidade.* Master Thesis, CPGCC-UFRS, Porto Alegre-RS, Brazil, 1995.

[SMV97] S.R.S. Souza, J.C. Maldonado, and S.R. Vergilio. Análise de mutantes e potenciais-usos: Uma avaliação empírica. In *VIII Cconferência Internacional de Tecnologia de Software: Qualidade de Software.* Curitiba-PR, June 1997. (in Portuguese).

[Sou96] S.R.S. Souza. *Avaliação do Custo e Eficácia do Critério Análise de Mutantes na Atividade de Teste de Programas.* Master Thesis, ICMSC/USP, São Carlos-Brazil, June 1996. (in Portuguese).

[Tai93] K.C. Tai. Predicate-based test 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.* Master Thesis - DCA/FEEC/Unicamp, Campinas - SP, Brazil, January 1992. (in Portuguese).

[Vin97] A.M.R. Vincenzi. *Subsídios para o Estabelecimento de Estratégias de Teste Baseadas na Técnica de Mutação.* Master Thesis, ICMSC-USP, São Carlos - SP, 1997. (in elaboration, in Portuguese).

[VMJ93a] S.R. Vergilio, J.C. Maldonado, and M. Jino. Influência do número de predicados na executabilidade de um caminho no contexto de teste baseado em fluxo de dados. In *XIX Conferencia Latinoamericana de Informatica.* Buenos Aires, Argentina, August 1993. (in Portuguese).

[VMJ93b] S.R. Vergilio, J.C. Maldonado, and M. Jino. Uma estratégia para geração de dados de teste. In *VII Simpósio Brasileiro de Engenharia de Software*, pages 306–319. Rio de Janeiro-RJ, Brazil, October 1993. (in Portuguese).

[VMJ95a] S.R. Vergilio, J.C. Maldonado, and M. Jino. *Estratégias de Geração de Dados de teste e a Satisfação de Critérios.* Technical Report - DCA/FEEC/Unicamp, Campinas - SP, Brazil, September 1995. (in Portuguese).

[VMJ95b] S.R. Vergilio, J.C. Maldonado, and M. Jino. Um experimento de aplicação de critérios baseados em fluxo de dados no teste de programas c. In *XV Congresso da Sociedade Brasileira de Computação e XXI Conferencia Latinoamericana de Informatica*, pages 941–952. Canela -RS, Brazil, August 1995. (in Portuguese).

[VMJ96a] S.R. Vergilio, J.C. Maldonado, and M. Jino. Infeasible paths within the context of data flow based criteria. In *VI International Conference on Software Quality*, pages 310–321. Ottawa-Canada, October 1996.

[VMJ96b] S.R. Vergilio, J.C. Maldonado, and M. Jino. *Resultados da Aplicação de Diferentes Técnicas de Geração de Dados de Teste Sensíveis a Erros.* Technical Report, DCA/FEEC/Unicamp, Campinas - SP, Brazil, 1996. (in Portuguese).

[VMJ96c] S.R. Vergilio, J.C. Maldonado, and M. Jino. *Uma Família de Critérios de Teste Baseados em Restrições.* Technical Report, DCA/FEEC/Unicamp, Campinas - SP, Brazil, 1996. (in Portuguese).

[VMJ97] S.R. Vergilio, J.C. Maldonado, and M. Jino. *Avaliação Empírica de Critérios Baseados em Restrições.* Technical Report, DCA/FEEC/Unicamp, Campinas - SP, Brazil, 1997. (in Portuguese).

[VMM91] J. Voas, L. Morell, and K. Miller. Predicting where faults can hide from testing. *IEEE Software*, Vol. SE-:41–47, March 1991.

[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.

[WJ91] E.J. Weyuker and B. Jeng. Analysing partition testing strategies. *IEEE Transactions on Software Engineering*, Vol. SE-17(7):703–711, July 1991.

[WMM94] W.E. Wong, A.P. Mathur, and J.C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In *Software Quality and Productivity - Theory, Practice, 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.

[YM89] D.F. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. *ACM Computer Surveys*, pages 48–54, 1989.

[ZAW92] J.S. Zeil, F.H. Affi, and L.J. White. Detection of linear errors via domain testing. *ACM Transactions on Software Engineering and Methodology*, Vol. 1(4):422–451, October 1992.

Apêndice A

Restrições para as Ferramentas Mothra e Proteum

Nesse apêndice, são apresentadas as tabelas contendo restrições de necessidade geradas para os operadores de mutação das ferramentas Mothra e Proteum.

As Tabelas A.1 e A.2 foram geradas por DeMilo e encontram-se em [DMO91]. As restrições mostradas são restrições de necessidade para os operadores de mutação da ferramenta Mothra. Entre elas estão restrições de predicado propostas por DeMilo para aproximar restrições de suficiência e garantir a avaliação incorreta de um predicado se ele contiver um erro.

As tabelas para os operadores de mutação da ferramenta Proteum (Tabelas A.3, A.4, A.6, A.5) foram derivadas durante a condução do experimento com a técnica Constraint Based Testing, descrito no Capítulo 3. Essas tabelas estão fortemente baseadas nas tabelas do sistema Mothra-Godzila. Elas contêm restrições de necessidade para revelar os erros descritos por esses operadores e podem ser utilizadas para a geração dos elementos requeridos pelos Critérios Restritos. Note que muitas da mutações em comandos, que incluem em geral, remoção dos mesmos, nãoestão associadas a nenhuma restrição de necessidade. Isso porque, muitas vezes, um estado intermediário incorreto é produzido apenas satisfazendo-se a restrição de alcançabilidade do comando mudado.

Tabela A.1: Restrições do Sistema Mothra-Godzila

Tipo	Descrição	Restrição
aar	array for array replacement	$A(e_1) \neq B(e_2)$
abs	absolute value insertion	$e_1 > 0, e_1 < 0, e_1 = 0$
acr	array constant replacement	$C \neq A(e_1)$
aor	arithmetic operator replacement	$(e_1 \sigma e_2) \neq (e_1 \phi e_2)$
		$(e_1 \sigma e_2) \neq (e_1)$
		$(e_1 \sigma e_2) \neq (e_2)$
		$(e_1 \sigma e_2) \neq Mod(e_1, e_2)$
asr	array for variable raplacement	$X \neq A(e_1)$
car	constant for array replacement	$A(e_1) \neq C$
cnr	comparable array replacement	$A(e_1) \neq B(e_2)$
csr	constant for scalar replacement	$X \neq C$
der	DO statement and replacement	$(e_2 - e_1) \geq 2$
		$e_2 \leq e_1$
lcr	logical connector replacement	$(e_1 \sigma e_2) \neq (e_1 \phi e_2)$
ror	relational operator replacement	$(e_1 \sigma e_2) \neq (e_1 \phi e_2)$
sar	scalar for array replacement	$A(e_1) \neq X$
scr	scalar for constant replacement	$C \neq X$
svr	scalar variable replacement	$X \neq y$

Tabela A.2: Outros Tipos de Mutações - Sistema Mothra-Godzila

Tipo	Descrição
crp	constant replacement
dsa	data statement alterations
grl	GOTO label replacement
rsr	RETURN statement replacement
san	statement analysis
sdl	statement deletion
src	source constant replacement
uoi	unary operator insertion

Tabela A.3: Mutação em Constantes

Tipo	Descrição	Restrições
cccr	constant for constant replacement	$C_1 \neq C_2$
ccsr	constant for scalar replacement	$X \neq C$
crcr	required constant replacement	$X \neq 0, X \neq -1, X \neq 1$

Tabela A.4: Mutação em Variáveis

Tipo	Descrição	Restrições
varr	mutate array reference	$A[e_1] \neq B[e_1]$
vdtr	domain traps	$X = 0, X < 0, X > 0$
		$A[e_1] > 0, A[e_1] < 0, A[e_1] = 0$
		$S.c = 0, S.c > 0, S.c < 0$
vprr	mutate pointer references	$p \neq q$
vsrr	mutate scalar references	$A[e_1] \neq X$, $S.c \neq X$
		$(*p) \neq X, Y \neq X$
vtrr	mutate structure references	$S \neq T$
vscr	structure component replacement	$S.c_1 \neq S.c_2$
vtwd	twiddle mutations	$PRED(x) \neq x, SUC(x) \neq x$

Tabela A.5: Mutações em Comandos

Tipo	Descrição	Restrições
sbrc	break replacement by continue	
sbrn	break out to nth level	
scrb	continue replacement by break	
scrn	continue out to nth level	
sdwd	do-while (CT) repalcement by while	CT=false
sglr	go to label replacement	
smvb	move brace up and down	
srsr	return replacement	
ssdl	statement deletion	
stri	trap on if (CT) condition	CT=false, CT=true
strp	trap on statement execution	
smtc	n-trip continue	
sswm	switch statement mutation	
swdd	while (CT) replacement by do-while	CT=false

Tabela A.6: Mutação em Operadores

Tipo	Descrição	Restrições
oaaa	arithmetic assignment mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oaan	arithmetic operator mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oaba	arithmetic assignement by bitwise assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oaea	arithmetic assignement by plain assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oaln	arithmetic operator by logical operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oarn	arithmetic operator by relational operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oasa	arithmetic assignement by shift assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oasn	arithmentic operator by shift operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obaa	bitwise assignement by arithmetic assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oban	bitwise operator by arithmetic assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obba	bitwise assignment mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obbn	bitwise operator mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obea	bitwise assignment by plain assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obln	bitwise operator by logical operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obng	bitwise negation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obrn	bitwise operator by relational operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obsa	bitwise assingmente by shift assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
obsn	bitwise operator by shift operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oeaa	plain assignment by arithmetic assignement	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oeba	plain assignment by bitwise assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oesa	palin assignment by shift assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oido	increment/decrement mutation	$(e_1 op_1) \neq (op_1 e_1)$, $(op_1 e_2) \neq (op_2 e_2)$ $(op_1 e_2) \neq (e_2 op_1)$, $e_1 \neq op e_2$
olan	logical operator by arithmetic operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
olbn	logical operator by bitwise operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
olln	logical operator mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
olng	logical negation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
orln	logical operator by relational operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
olsn	logical operator by shift operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
oran	relational operator by arithmetic operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
orbn	relational operator by bitwise operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
orln	relational operator by logical operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
orrn	relational operator mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
orsn	relational operator by shift operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
osaa	shift assignment by arithmetic assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
osan	shift operator by arithmetic operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
osba	shift assignment by bitwise assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
osbn	shift operator by bitwise operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
osea	shift assignment by plain assignment	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
osln	shift operator by logical operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
osrn	shift operator by relational operator	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
ossn	shift operator mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$
ossa	shift assignment mutation	$(e_1 op_1 e_2) \neq (e_1 op_2 e_2)$

Apêndice B

Propriedades dos Critérios Baseados em Restrição

Nesse apêndice, são apresentadas as provas relacionadas às propriedades dos critérios Restritos, discutidas no Capítulo 3: análise de inclusão e análise de complexidade.

Como foi dito no Capítulo 3, consideranto a restrição C=true, o critério restrito inclui o critério estrutural correspondente, mesmo na presença de não executabilidade. Se diferentes conjuntos de restrições forem escolhidos os Critérios Restritos podem ser incomparáveis entre si. Se um conjunto comum de restrições é associado a cada elemento requerido, para todos os critérios estruturais, têm-se:

- *(todos-caminhos-restritos)* $\Rightarrow$ *(todos-potenciais-du-caminhos-restritos)*: o conjunto de potenciais-du-caminhos associados às restrições de C são cobertos por um subconjunto de caminhos do programa que os inclui e que estão associados às mesmas restrições. Portanto, um subconjunto dos elementos requeridos pelo critério todos-caminhos-restritos.

- *(todos-potenciais-du-caminhos-restritos)* $\Rightarrow$ *(todos-du-caminhos-restritos)*: visto que todo du-caminho é um potencial-du-caminho [Mal91], um subconjunto de potenciais-du-caminhos associados às restrições de C cobre os du-caminhos associados às mesmas restrições e requeridos pelo critério todos-du-caminhos-restritos.

- *(todos-potenciais-du-caminhos-restritos)* $\Rightarrow$ *(todos-potenciais-usos/du-restritos)*: um subconjunto de potenciais-du-caminhos associados às restrições de C cobre as associações associadas às mesmas restrições e requeridas pelo critério todos-potenciais-usos/du

- *(todos-potenciais-usos/du-restritos)* $\Rightarrow$ *(todos-potenciais-usos-restritos)*: um mesmo subconjunto de potenciais-du-caminhos associados às restrições de C cobre as associações associadas às mesmas restrições e requeridas pelo critério todos-potenciais-usos/du e pelo critério todos-potenciais-usos-restritos.

- *(todos-du-caminhos-restritos)* $\Rightarrow$ *(todos-usos-restritos)*: para cada uso, um du-caminho

é derivado, cobrir todos-du-caminhos e satisfazer o conjunto C de restrições, implica em cobrir todos-usos associados ao mesmo conjunto R.

• *(todos-potenciais-usos-restritos)* $\Rightarrow$ *(todos-usos-restritos)*: visto que toda associação é uma potencial-associação [Mal91], então, se um mesmo conjunto C de restrições é derivado para cada uma das associações, uma associação restrita também será uma potencial-associação restrita. Todos potenciais-usos restritos inclui todos-usos restritos.

• *(todos-usos-restritos)* $\Rightarrow$ *(todos-ramos-restritos)*: Se todo ramos é coberto por um caminho que cobre uma associação, visto que todos-usos inclui todos-ramos, e para cada associação está associado o conjunto de restrições R. Então cada arco restrito também será coberto.

• *(todos-ramos-restritos)* $\Rightarrow$ *(todos-nos-restritos)*: Se todo nó é coberto por um ramo, e todo ramo está associado as mesmas restrições de R, então todo nó restrito é coberto por um ramo restrito.

Na presença de não executabilidade, essa ordem é alterada. Abaixo estão as relações para os Critério Executáveis (ou *), considerando que os programas satisfazem a seguinte propriedade: possuem pelo menos uma definição de variável no nó de entrada do programa.

• $(todos - caminhos - restritos)^* \Rightarrow (todos - potenciais - du - caminhos - restritos)^*$: para todo potencial-du-caminho restrito executável existe um caminho completo executável que o cobre e que satisfaz o conjunto R. Esse caminho é requerido pelo critério $(todos - potenciais - du - caminhos - restritos)^*$.

• $(todos - potenciais - du - caminhos - restritos)^* \Rightarrow (todos - du - caminhos - restritos)^*$: se todo du-caminho é um potencial-du-caminho, considerando o mesmo conjunto de restrições R, todo du-caminho-restrito é um potencial-du-caminho-restrito. Portanto, se o du-caminho restrito é executável existirá o potencial-du-caminho-restrito executável correspondente.

• $(todos - potenciais - du - caminhos - restritos)^* \Rightarrow (todos - potenciais - usos/du - restritos)^*$: um subconjunto de potenciais-du-caminhos restritos executáveis satisfaz o critério $(todos - potenciais - usos/du - restritos)^*$.

• $(todos - caminhos - restritos)^* \Rightarrow (todos - potenciais - usos - restritos)^*$: um subconjunto de caminhos restritos executáveis satisfaz o critério $(todos - potenciais - usos - restritos)^*$.

• $(todos - potenciais - usos - restritos)^* \Rightarrow (todos - usos - restritos)^*$: visto que toda associação é uma potencial-associação [Mal91], então, se um mesmo conjunto C de restrições é derivado para cada uma das associações, toda associação restrita é uma potencial-associação restrita. Se a associação restrita for executável a potencial-associação restrita também o será.

• $(todos - potenciais - usos/du - restritos)^* \Rightarrow (todos - ramos - restritos)^*$: para todo ramo restrito executável, existe um caminho completo restrito executável que o cobre. Como existe pelo menos uma definição de variável no nó de entrada e $(todos - potenciais - usos/du)^*$

$\Rightarrow$ $(todos - ramos)^*$, existirá um potencial-du-caminho incluído nesse caminho completo. Se o caminho completo restrito é executável, o potencial-du-caminho restrito nele incluído também o será e cobrirá portanto o ramo-restrito.

• $(todos - potenciais - usos - restritos)^* \Rightarrow (todos - ramos - restritos)^*$: para todo ramo restrito executável (R,E) existe um caminho completo restrito executável que o cobre. Como existe pelo menos uma definição de variável no nó de entrada e $(todos - potenciais - usos)^* \Rightarrow (todos - ramos)^*$, existe uma associação restrita executável, que é coberta por esse caminho completo e que cobre o ramo restrito.

• $(todos - ramos - restritos)^* \Rightarrow (todos - nos - restritos)^*$: os nós restritos serão cobertos por um conjunto de caminhos que também satisfazem as restrições de R, para se chegar ao nó j requerido, existirá um arco restrito (i,j) executável, que está incluído no caminho completo.

Nos casos abaixo, a inclusão não acontece nem mesmo para os critérios estruturais, independentemente das restrições estabelecidas [Mal91].

• $(todos-potenciais-du-caminhos-restritos)^* \not\Rightarrow (todos-potenciais-usos-restritos)^*$;

• $(todos - potenciais - usos/du - restritos)^* \not\Rightarrow (todos - potenciais - usos - restritos)^*$;

• $(todos - du - caminhos - restritos)^* \not\Rightarrow (todos - usos - restritos)^*$;

• $(todos - usos - restritos)^* \not\Rightarrow (todos - ramos - restritos)^*$.

Com relação à complexidade dos Critérios Restritos, foi visto que ela dependerá do nível e da técnica aplicados para gerar as restrições.

Nível 2: aplicando-se a técnica Variable Mutation, o pior caso é o caso no qual as k variáveis da associação $(i,(j,k),\{v_1,v_2,..,v_k\})$, forem do mesmo tipo e valores diferentes serão exigidos para combinações dois a dois destas variáveis. O número de restrições é dado por $(k-1)+(K-2)...+2+1 = k(k+1)/2$.

Apêndice C

Aspectos de Implementação

Este apêndice apresenta aspectos de implementação dos módulos propostos no Capítulo 4, como extensão à ferramenta POKE-TOOL para apoiar a utilização dos Critérios Restritos e facilitar a geração de dados de teste. Primeiramente, serão fornecidas as descrições e modos de utilização dos "scripts" da versão SUN-UNIX da POKE-TOOL e que ativam os módulos *Pokepadrao*, *Pokepaths* e *Pokeheuris*. Os scripts correspondentes aos módulos *Cb-aval* e *Cb-elem* são então apresentados e ao final apresentam-se os arquivos gerados pelo módulo *Cb-aval* implementado para o program *cal*, utilizado no experimento descrito no Capítulo 5.

C.1 Descrição dos "Scripts" Correspondentes aos Novos Módulos da POKE-TOOL

C.1.1 Eliminação de Padrões/Elementos não Executáveis

Este "script' permite o tratamento de padrões de não executabilidade através das operações de inserção, eliminação , habilitação e desabilitação de padrões. Essas operações são feitas em um arquivo $< funcao >$ *.padrao*, o qual contém, para cada função em teste, os padrões de não executabilidade por critério e os elementos requeridos pelo critério associado ao padrão. Cada operação deve repercutir de forma coerente nos arquivos de elementos não executados para a função e critério. Também é mantido um arquivo com os elementos não executáveis.

```
pokepadrao -d<funcao> --[i|r|e|d][nos|-arcs|-pu|-pudu|-p] "<padrao>"|-a|x to y
```

<funcao>: nome do diretório onde estão os arquivos pduoutput, puoutput, pdupaths para o programa que contém o padrão.

-i: insere padrão

-r: remove padrão

-e: habilita padrão durante a avaliação

-d: desabilita padrão durante a avaliação

-nos: critério todos-nós

-arc: critério todos-arcos

-pu: critério todos-potenciais-usos

-pudu: critério todos-potenciais-usos/du

-pdu: critério todos potenciais-du-caminhos

<padrao< : padrão a ser utilizado na busca, pode ser do tipo:

1. nó "n"
2. arco "j k" ou "(j,k)"
3. associação "(i,(j,k),{x})"

   sendo que {} indicará c.r.a qq x.
4. associação "(i,(j,k),{x})/du"
5. seqüência de nós contínua: "n1 n2 nn"
6. seqüência de nós não contínua: " $n_1 n_m ... n_{n-1} n_n$"

-a: usar todos os padrões contidos no arquivo <funcao>.padrao para o critério

-x to y: usar o intervalo de padrões de número x a número y para o critério

Exemplo de Utilização: pokepadrao -dsort -i -pdu "1 3 ...7 8"

Insere o padrão no arquivo *sort.padrao* do tipo seqüência de nós não contínua para o critério todos-potenciais-du-caminhos. Os elementos que incluírem esse padrão são eliminados do arquivo *puoutput.tes* e inseridos no arquivo *pdune.tes*. Uma nova cobertura é fornecida.

C.1.2 Geração de um Conjunto de Caminhos Completos

```
pokepaths -d<funcao> -[nos|-arcs|-pu|-pudu|-pdu] ["elemento"|-a|x to y] -c<estrat>
```

<funcao>: nome do diretório onde estão os arquivos pduoutput, puoutput, pdupaths

-nos: critério todos-nós

-arc: critério todos-arcos

-pu: critério todos-potenciais-usos

-pudu: critério todos-potenciais-usos/du

-pdu: critério todos potenciais-du-caminhos

"elemento": o elemento a ser coberto, segue os mesmos formatos e tipos de padrão.

-a: todos os elementos requeridos para o critério

x to y: número dos elementos (ou intervalo de) a ser considerado na geração.

-c<estrat>: indica a estratégia de seleção dos caminhos, correspondente:

1: menor número de predicados

2: menor número de nós

3. maior número de predicados

4. maior número de nós

5. caminho típico (o mais frequentemente executado)

6. caminho menos executado

7. maior testabilidade

8. menor testabilidade

9. o de maior complexidade etc.

Exemplo de Utilização: pokepaths -dfind -pu "(1,(4,6),{util})" -c1

O módulo pokepaths fornecerá uma lista de caminhos completos candidatos a cobrir a associação não contínua, ou seja, livres de definição c.r. à *util*, ou seja que cobrem a associação fornecida. Esses caminhos são selecionados de acordo com a estratégia menor número de predicados.

C.1.3 Aplicação de Heurísticas e Execução Simbólica

```
pokeheuris -dfuncao -[nos|-arcs|-pu|-pudu|-pdu] ["elemento"| x to y] [-s|n]
```

<funcao>: nome do diretório onde estão os arquivos pduoutput, puoutput, pdupaths

-nos: critério todos-nós

-arc: critério todos-arcos

-pu: critério todos-potenciais-usos

-pudu: critério todos-potenciais-usos/du

-pdu: critério todos potenciais-du-caminhos

"elemento": segue o mesmos formatos especificados para pokepaths

x to y: número dos elementos (ou intervalo de) a ser considerado na aplicação da heurística

-s||n ativa ou não o módulo pokepadrao

Durante a aplicação da heurística, novos padrões podem ser determinados. Portanto uma atualização dos arquivos $< funcao >$ *.padrao* e *criteriooutput.tes* será realizada. O novo padrão é inserido e os elementos correspondentes eliminados.

Exemplo de Utilização: pokeheuris -dsort -i -pdu "(1,(1,6),{n})"

O módulo *Pokeheuris* determinará se a associação fornecida requerida para a função *sort* é ou não executável. Se ela for não executável um novo padrão está disponível. A opção -s ativa o módulo *Pokepadrao*. O novo padrão é inserido em *sort.padrao*. Os elementos requeridos pelo critério todos-potenciais-du-caminhos que contiverem o novo padrão são eliminados e uma nova análise de cobertura é realizada

C.2 "Scripts" que Apoiam a Utilização dos Critérios Restritos

Abaixo são apresentados os scripts *Cb-aval e Cb-auxaval* que implementam o módulo *Cb-aval* e parte do script *Cb-elem* implementada.

```
---------------------- cb-aval -------------------
#!/bin/sh

#############################################################################
#      Ja' foram criados os arquivos cb-elementos, (cb-assoc.tes,           #
#      cb-pudesc.tes e cb-desc.tes), no diretorio cb-nomefuncao             #
#       Essa versao e' para o pokebatch (pokemal e pokeauxmal), que         #
#      comprime os pathI.tes                                                #
#############################################################################
```

```
case $# in
0) echo 'Uso: cbavalcomp -d<diretorio funcao> -pdu|-pu|-pudu|-nos|-arcs  [x to y]'; exit 2;;
1) echo 'Uso: cbaval -d<diretorio funcao> -pdu|-pu|-pudu|-nos|-arcs  [x to y]';
 echo 'Numero incorreto de parametros'; exit 2;;
2) dir=$1; criterio=$2; beg=1; last=0; end=`expr $last + 1`;;
# lembrar que last deve ser o numero de casos de teste existente.
3) dir=$1; criterio=$2; beg=$3; last=$3; end=`expr $3 + 1`;;
4) dir=$1; criterio=$2; beg=$3; last=$3; end=`expr $3 + 1`;;
5) dir=$1; criterio=$2; beg=$3; last=$5; end=`expr $5 + 1`;;
esac
# testa valores iniciais e finais de caso de testes
if test "$beg" -gt "$end"
then
    exit 2
fi
oldir=`pwd`
# retira -d de nome de diretorio
dir=`echo $dir|sed 's/^-d//g'`
if test -d cb-$dir
then
     :
else
  echo "Nao encontrei o diretorio cb-$dir"
  exit 2
fi
#  pode-se pensar em uma opcao -all, para todos os criterios
case $criterio in
"-pdu") criterio=1; cri="pdu"; cp $dir/pdupaths.tes  cb-$dir/pdupaths.tes;
 arquivo="cb-paths.tes";;
"-pu") criterio=2; cri="pu"; cp $dir/puassoc.tes  cb-$dir/puassoc.tes;
arquivo="cb-assoc.tes";;
"-pudu") criterio=3; cri="pudu"; cp $dir/puassoc.tes  cb-$dir/puassoc.tes;
arquivo="cb-assoc.tes";;
"-arcs") criterio=4; cri="arcs";cp $dir/arcprim.tes  cb-$dir/arcprim.tes;
arquivo="cb-arcs.tes";;
"-nos") criterio=5; cri="nos";cp $dir/nos_grf.tes  cb-$dir/nos_grf.tes;
arquivo="cb-nos.tes";;
*) echo "Erro: criterio \"$criterio\" nao esta disponivel !" 1>&2 ;
   cd $oldir;
   exit 2;;
esac
#copia descritores
cp $dir/des_$cri.tes cb-$dir/des_$cri.tes
```

```
#copiar programa instrumentado
cp testeprg.c cb-$dir/testeprg.c
num_linhas=`wc -l testeprg.c | awk '{print $1}'`
# muda para diretorio cb, se o dir for um path longo cb-path nao faz sentido
cd cb-$dir
# se existe um arquivo cbcriteriohis.tes
# na verdade testara' depois. Sempre vai acumulando... pode-se colocar uma
# opcao para que se recomece a avaliacao, e se destrua cb-his
if test -d good
then
  :
else
  mkdir good
fi
while test "$beg" != "$end"
do
   uncompress ../$dir/path$beg.tes
   if test -f ../$dir/path$beg.tes
   then
     cp ../$dir/path$beg.tes path.tes
     compress ../$dir/path$beg.tes
   else
     echo "Nao encontrei o arquivo path$beg.tes"
   fi
   if test -f ${cri}his.tes
   then
     cp ${cri}his.tes ${cri}his.bak
   else
     echo  > ${cri}his.bak
   fi
   if test -f cb-${cri}his.tes
   then
     :
   else
     echo > cb-${cri}his.tes
   fi
   if test -f cb-${cri}exe.tes
   then
     :
    else
      echo "          CB-ASSOCIACOES EXECUTADAS     " > cb-${cri}exe.tes
      echo "***         Cobertura  Total = 0.000000 " >> cb-${cri}exe.tes
    fi
    echo "Avaliando caso de teste numero $beg "
```

```
 echo " 0" >> path.tes
 if avaliador $criterio
 then
   # At least one test case was successfull
   :
 else
   # This test case failed
   rm -f avalog.tes # Clean up intermediate files
   echo ''
   echo "Mensagem: avaliacao falhou para o caso de teste $beg"
   exit 2
 fi
if diff ${cri}his.bak ${cri}his.tes > difere
then
  echo "Caso de teste $beg nao matou nenhum elemento"
else
  cat difere | grep ">" | awk '{printf " %s\n",$2}' > good_aux
fi
# para cada elemento em good_aux verificar as condicoes nos arquivos cb
# obtem numero de elementos de good_aux
if test -f good_aux
then
  number=`wc -l good_aux | awk '{print $1}'`
  number=`expr $number + 1`
  i=1
  echo > temp_assoc
  while test "$i" != "$number"
  do
   # j = elementos (numero da associacao morta) de good
   j=`head -$i good_aux | tail -1`
   # obtem todas as cb a partir de j, j ja' possui um branco
   grep "$j-"  $arquivo > assoc
   # numero de cb-associacoes derivadas para a associacao j
   qtas_assoc=`wc -l assoc | awk '{print $1}'`
   qtas_assoc=`expr $qtas_assoc + 1`
   k=1
   while test "$k" != "$qtas_assoc"
   do
     # linha contem a linha do arquivo assoc
     elem=`head -$k assoc | tail -1 | awk '{print $1}'`
     # em temp-assoc estao as associacoes derivadas de k e que nao estao em
     # cb-puhis, para todo k do caso de teste beg
     if grep " $elem" cb-${cri}his.tes >> temp
     then
```

```
          :
        else
          echo " $elem" >> temp_assoc
        fi
        k='expr $k + 1'
      done
      i='expr $i + 1'
    done
   cb-auxaval $cri $num_linhas $beg $arquivo $dir
  fi
  rm -f good_aux difere assoc temp_assoc temp
 beg='expr $beg + 1'
# compress ../$dir/path$beg.tes
done
sort -u cb-${cri}exe.tes -ocb-${cri}exe.tes
ntotal='tail -1 cb-rel${cri}.tes | awk '{print $5}''
# calcula cobertura
n='wc -l cb-${cri}exe.tes | awk '{print $1}''
head -'expr $n - 1'  cb-${cri}exe.tes  > temp
# O programa executavel calcula-cob, calcula a cobertura e
# a escreve no arquivo cobertura.tes
n='expr $n - 2'
calcula-cob cobertura.tes $n  $ntotal
# substitui no arquivo
cat temp cobertura.tes > cb-${cri}exe.tes
# Cria arquivo de associacoes nao executadas
echo " CB-ASSOCIACOES NAO EXECUTADAS " > cb
sort -u $arquivo -ocb-temp
comm -13 cb-${cri}exe.tes cb-temp | grep -v "A" >> cb
cat cb cobertura.tes >  cb-${cri}output.tes
cat cb-${cri}output.tes
rm -f temp cobertura.tes good_aux cb-temp cb testeprg.c
# volta ao diretorio inicial
cd $oldir
exit 0
```

---------- cb-auxaval ----------

```
#!/bin/sh
# Executa os programas com as condicoes necessarias para matar as
# associacoes indicadas no arquivo temp_assoc e coloca as associacoes
# mortas em cb-criteriohis.tes
criterio=$1
```

```
num_linhas=$2
casoteste=$3
arq_elem=$4
func=$5
echo " * * Avaliando as restricoes para o dado de teste * *"
echo ""
# Obtem a entrada para executar o programa correspondente ao casoteste
echo > vazio
if test -d ../input
then
  if test -d ../keyboard
    then
      keyboard="keyboard/keyboard$casoteste.tes"
      read input < ../input/input$casoteste.tes
    else
      keyboard="`pwd`/vazio"
      read input < ../input/input$casoteste.tes
  fi
else
  if test -d ../keyboard
  then
    keyboard="keyboard/keyboard$casoteste.tes"
    input=""
   else
     keyboard="`pwd`/vazio"
     input=""
   fi
fi
echo > good_temp
echo > nogood
#numero de cb-associacoes candidatas
num_assoc=`wc -l temp_assoc | awk '{print $1}'`
num_assoc=`expr $num_assoc + 1`
k=2    # porque temp-assoc tem ao menos um linha em branco
while test "$k" != "$num_assoc"
do
   elem=`head -$k temp_assoc | tail -1 | awk '{print $1}'`
   if grep " $elem"  good_temp nogood > his.tes
   then
     :
     # ja' foi coberta
   else
     # pegar linha com esse elemento em cb-relcriterio
      linha=`grep " $elem" cb-rel${criterio}.tes`
```

```
if test "$linha" != ""
then
  numcond=`echo $linha | awk '{print $1}'`
  grupo=`echo $linha | sed "s/^$numcond//g"`
  # de posse do numero do arco/restricao (numcond)
  # Procurar posicao de insercao em testeprg.
  cond=`head -$numcond cb-arcs.tes | tail -1`
  ini=`echo $cond | awk '{printf "%s %s ", $1, $2}'`
  condicao=`echo "$cond" | sed "s/^$ini//g"`
  # echo "/* $func-$numcond "
  pos=`grep -n "/* $func-$numcond " testeprg.c | awk '{print $1}'`
  pos=`echo $pos | sed "y/:/ /"`
  # insere condicao na posicao indicada
  posmenos=`expr $num_linhas - $pos`
  # divide arquivo testeprog.c em  dois a partir de pos
  # e insere condicao gerada
  head -$pos testeprg.c > cb-exe.c
  echo "if" "$condicao" '{FILE * f=fopen("s.tes","w");
            fprintf(f,"satisfez"); fclose(f);
             exit(1);}' >> cb-exe.c
  tail -$posmenos testeprg.c >> cb-exe.c
  dir=`pwd`
  cp cb-exe.c ..
  cd ..
  if gcc cb-exe.c -o cb-exe > out.tes
  then
    # compilou corretamente
    # colocar o timer  se necessario
    cb-exe $input < $keyboard > out.tes
    if test -f s.tes
    then
      # satisfez a restricao e gerou o arquivo s.tes
      arquivo="good_temp"
      rm -f s.tes
    else
      :
       # nao satisfez a condicao ou deu timeout
       arquivo="nogood"
    fi
    cd $dir
    for assoc in `echo $grupo | sed 's/-^//g'`
    do
      elem=`grep " $assoc" temp_assoc | awk '{print $1}'`
```

```
            if test "$elem" != ""
            then
              echo " $elem" >> $arquivo
            fi
          done
        else
          echo "Nao consegui gerar programa executavel para o cb-avaliador"
          exit 2
        fi
        rm -f out.tes
        cd $dir
      else
        # associacao nao possui restricao associada
        echo " $elem" >> good_temp
      fi
    fi
    k=`expr $k + 1`
done
if test -f good_temp
then
  l=`wc -l good_temp | awk '{print $1}'`
  # o cb-good esta' em good-temp.
  if test "$l" != "1"
  then
    cd good
    cp ../good_temp cb-good${criterio}-$casoteste
    sort -u cb-good${criterio}-$casoteste -ocb-good${criterio}-$casoteste
    cd ..
    #atualiza cb-his a partir de goodtemp
    sort cb-puhis.tes -ohis.tes
    sort -m good_temp his.tes -ocb-puhis.tes
    sort -u cb-puhis.tes -ocb-puhis.tes
    as=`cat good_temp`
    # echo $as
    for a in `echo $as | sed 's/-^//g'`
    do
      grep " $a" $arq_elem >> cb-${criterio}exe.tes
    done
  fi
fi
# atualizar puhis.tes
cp puhis.tes his.tes
rm -f puhis.bak
num_his=`wc -l his.tes | awk '{print $1}'`
```

```
num_his=`expr $num_his + 1`
echo > puhis.tes
k=1
while test "$k" != "$num_his"
do
   elem=`head -$k his.tes | tail -1 | awk '{print $1}'`
   if grep " $elem" good_aux > out
   then
     # echo "achei good_aux $elem"
     if grep " $elem-" nogood > out
     then
        : #  echo "achei nogood $elem"
     else
       echo "$elem" >> puhis.tes
       # echo "achei good_aux e nao nogood $elem"
     fi
   else
     # echo " nao achei good_aux $elem"
     echo "$elem" >> puhis.tes
   fi
   k=`expr $k + 1`
done
rm -f out cb-exe.c cb-exe nogood his.tes vazio good_temp
exit 0
```

---------- cb-elem ----------

```
#!/bin/sh
#############################################################################
#  Tem como entrada o arquivo cb-arcs.tes, mais o arquivo puassoc.tes  #
#  cb-arcs.tes possui um arquivo de arcos requeridos e restricoes      #
#   associadas. A posicao que ela devera' ser inserida no arquivo      #
#  testeprg.c e' dada pelo comentario /* numero do cb-arco */          #
#  Supoe-se que cb-arcs.tes esta no diretorio da cb-funcao, que foram  #
#  gerados utilizando-se uma opcao no script poketool                  #
#  A partir deles, geracb gerara' o arquivo cb-assoc.tes  de elementos #
#  requeridos + restricoes.  Tambem o arquivo cb-purel.tes, que        #
#  relaciona as associacoes e posicoes de insercao/ restricoes         #
#  Geracb funciona so' para o criterio pu, tem que ser estendido para  #
#  funcionar para qualquer criterio dado pelo parametro criterio       #
#############################################################################
```

```
# OBS: quando cb-arcs.tes e' vazio, ou seja wc -l =0, nao existem
# restricoes associadas aos arcos. Entao cb-assoc sera' igual a
# puassoc. Se puassoc e' vazio, cb-assoc tambem sera' (ou seja contera
# apenas a linha de cabecalho.

dir=$1
criterio=$2
if test -d cb-$dir
then
  :
else
  echo "Nao existe diretorio preparado para a funcao $dir"
fi
if test -f cb-$dir/cb-arcs.tes
then
  :
else
  echo "Nao encontrei as restricoes correspondentes aos arcos: cb-arcs.tes"
  cd $oldir
  exit 2
fi
case $criterio in
"-pdu") cri="pdu"; cp $dir/pdupaths.tes  cb-$dir/pdupaths.tes; arquivo="cb-paths.tes";;
"-pu") cri="pu"; cp $dir/puassoc.tes  cb-$dir/puassoc.tes; arquivo="cb-assoc.tes";;
"-pudu") cri="pudu"; cp $dir/puassoc.tes  cb-$dir/puassoc.tes; arquivo="cb-assoc.tes";;
"-arcs") cri="arcs";cp $dir/arcprim.tes  cb-$dir/arcprim.tes; arquivo="cb-arcs.tes";;
"-nos") cri="nos";cp $dir/nos_grf.tes  cb-$dir/nos_grf.tes; arquivo="cb-nos.tes";;
*) echo "Erro: criterio \"$criterio\" nao esta disponivel !" 1>&2 ;
   cd $oldir; exit 2;;
esac
oldir=`pwd`
cd cb-$dir
# numero de elementos requeridos pelo criterio
qtos=`wc -l puassoc.tes | awk '{print $1}'`
qtos=`expr $qtos - 1`
# assoc_temp nao tem cabecalho
tail -$qtos puassoc.tes > assoc_temp
echo "ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS CB-POT-USOS E
      CB-POT-USOS/DU" > cb-assoc.tes
rm -f temp_desc
echo > arcs
i=1
while test "$qtos" != "0"
```

```
do
  qtos=`expr $qtos - 1`
  #pegar linha do arquivo e dividir arquivo em  dois
  assoc=`head -1 assoc_temp`
  tail -$qtos assoc_temp > assoc_t
  mv assoc_t assoc_temp
  if echo "$assoc" | grep "$i) <" > arcs
  then
    # gerar restricoes para a associacao i
    # eliminar o numero da associacao do string assoc
    assoc=`echo $assoc | sed "s/^$i)//g"`
    # recuperar arco da associacao
    arco=`echo "$assoc" | awk '{print $1}' | sed y/","/" "/ |
          awk '{printf "%s,%s", $2, $3}'`
    # escrever associacao no arquivo / associacao sem restricao
    echo " $i-1)" "$assoc" >> cb-assoc.tes
    echo  "$i-1) 0 " >> temp_desc
    # pesquisar por arco em arquivo cb-arcs.tes
    if grep $arco cb-arcs.tes > arcs
    then
      number=`wc -l arcs | awk '{print $1}'`
      number=`expr $number + 1`
      j=1
      while test "$j" != "$number"
      do
        linha=`head -$j arcs | tail -1`
        ini=`echo $linha | awk '{printf "%s %s ", $1, $2}'`
        n=`echo $linha | awk '{print $1}'`
        cond_arco=`echo "$linha" | sed "s/^$ini//g"`
        jj=`expr $j + 1`
        echo " $i-$jj) $assoc" "$cond_arco" >> cb-assoc.tes
        echo  "$i-$jj) $n " >> temp_desc
        j=`expr $j + 1`
      done
    fi
    i=`expr $i + 1`
  else
    # linha contem apenas comentario, repeti-la
    echo $assoc >> cb-assoc.tes
  fi
done
rm -f cb-relpu.tes
# a partir de  temp_desc gerar o descritor para o criterio.
if test -f temp_desc
```

```
then
 # number e' o numero de condicoes existentes
 number='wc -l cb-arcs.tes | awk '{print $1}''
 number='expr $number + 1'
 i=1
 while test "$i" != "$number"
 do
   grupo='grep " $i " temp_desc | awk '{print $1}''
   as=""
   for a in 'echo $grupo | sed 's/-^//g''
   do
     as="$as $a"
   done
   if test "$as" != ""
   then
     echo " $i  $as" >> cb-relpu.tes
   fi
   i='expr $i + 1'
 done
else
  echo > cb-relpu.tes
fi
n='wc -l temp_desc | awk '{print $1}''
echo "Numero de cb-puassociacoes = $n" >> cb-relpu.tes
rm -f assoc_temp temp_desc arcs
cd $oldir
exit 0
```

C.3 Arquivos Gerados pelo Cb-aval

Esta seção apresenta os arquivos gerados pelo módulo *Cb-aval* cujo os aspectos de implementação foram discutidos no Capítulo 4. Será utilizado para ilustração o programa *cal* do Unix usado no experimento descrito no Capítulo 5. Abaixo é apresentada a descrição funcional do programa.

```
cal(1)                    User Commands                    cal(1)
NAME
     cal - display a calendar
SYNOPSIS
     cal [ [ month ] year ]
AVAILABILITY
     SUNWesu
DESCRIPTION
     The cal utility writes a Gregorian calendar to standard out-
     put.  If the year operand is specified, a calendar for that
     year is written.  If no operands are specified, a calendar
     for the current month is written.
OPERANDS
     The following operands are supported:
     month   Specify the month to be displayed, represented as a
             decimal integer from 1 (January) to 12 (December).
             The default is the current month.
     year    Specify the year for which the calendar is
             displayed, represented as a decimal integer from 1
             to 9999.  The default is the current year.
ENVIRONMENT
     See environ(5) for descriptions of the following environment
     variables that affect the execution of cal:  LC_TIME,
     LC_MESSAGES, and NLSPATH.
EXIT STATUS
     The following exit values are returned:
     0         Successful completion.
     >0        An error occurred.
SunOS 5.5.1          Last change: 1 Feb 1995
```

Serão utilizadas apenas as funções menores *jan1* e *pstr* do programa *cal*, para ilustrar os arquivos gerados por *Cb-elemen* e *Cb-aval*. Mas esses arquivos foram gerados para as outras funções do programa. Primeiramente são fornecidos os grafos de fluxo de controle para estas funções (Figuras C.1 e C.2). Os comandos associados a cada nó, podem ser vistos no código correspondente a cada função, apresentado abaixo de cada grafo.

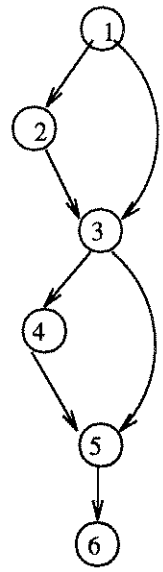


Figura C.1: Grafo de Fluxo de Controle para a Função *jan1*

```
---------------- Arquivo testeprog.c --------------------

jan1(int yr)
/*  1 */        {
                   FILE * path = fopen("jan1/path.tes","a");
                   static int printed_nodes = 0;
/*  1 */           register y, d;
                   ponta_de_prova(1);
/*  1 */           y = yr;
/*  1 */           d = 4+y+(y+3)/4;
/*  1 */           if(y > 1800)
/*  2 */           {
                      ponta_de_prova(2);
/*  2 */              d -= (y-1701)/100;
/*  2 */              d += (y-1601)/400;
/*  2 */           }
                   ponta_de_prova(3);
/*  3 */           if(y > 1752)
/*  4 */           {
                      ponta_de_prova(4);
/*  4 */              d += 3;
/*  4 */           }
                   ponta_de_prova(5); ponta_de_prova(6);  fclose(path);
/*  5 */           return(d%7);
/*  6 */        }
```

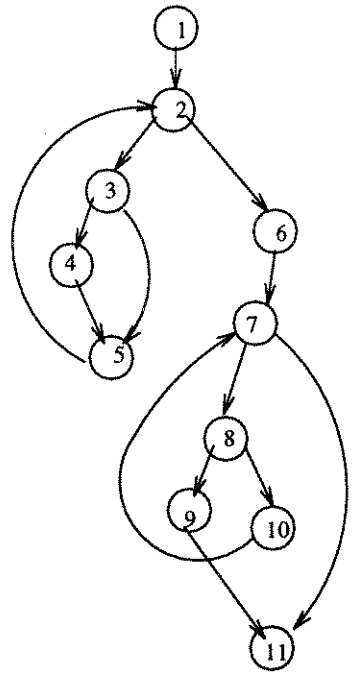


Figura C.2: Grafo de Fluxo de Controle para a Função *pstr*

```
pstr(char *str, int n)
/*  1 */        {
                   FILE * path = fopen("pstr/path.tes","a");
                   static int printed_nodes = 0;
/*  1 */           register i;
/*  1 */           register char *s;
                   ponta_de_prova(1);
/*  1 */           s = str;
/*  1 */           i = n;
/*  2 */           while(i--)
/*  3 */           {
                      ponta_de_prova(2);
                      ponta_de_prova(3);
/*  3 */              if(*s++ == '\0')
/*  4 */              {
                         ponta_de_prova(4);
/*  4 */                 s[-1] = ' ';
/*  4 */              }
                      ponta_de_prova(5);
/*  5 */           }
                   ponta_de_prova(2);
```

```
                    ponta_de_prova(6);
/*  6 */            i = n+1;
/*  7 */            while(i--)
/*  8 */            {
                       ponta_de_prova(7);
                       ponta_de_prova(8);
/*  8 */               if(*--s != ' ')
/*  9 */               {
                          ponta_de_prova(9);
/*  9 */                  goto out0_break;
/*  9 */               }
                       ponta_de_prova(10);
/* 10 */            }
                    ponta_de_prova(7);
                    out0_break:
                    ponta_de_prova(11);
/* 11 */            s[1] = '\0';
/* 11 */            printf("%s\n", str);
                    fclose(path);
/* 11 */        }
```

C.3.1 Programa instrumentado gerado para o Cb-aval

O programa instrumentado *cb-testeprog.c* é o programa utilizado pelo módulo *Cb-aval* para verificar se as restrições foram ou não satisfeitas. As restrições são inseridas nos pontos de inserção de restrições especificados pelos comentários /* função-número */ (Ex: /* pstr-5 */).

---------- Programa Instrumentado - cb-testeprog.c ----------

```
pstr(char *str, int n)
{
register i;
register char *s;
s = str;
i = n;
while(i--)
  {
      /* pstr-1 */ /* pstr-2 */
if(*s++ == '\0')
  {
```

```
    /* pstr-4 */
s[-1] = ' ';
  }
                /* pstr-5 */ /* pstr-6 */
  }
/* pstr-3 */
i = n+1;
while(i--)
  {
        /* pstr-7 */ /* pstr-8 */
if(*--s != ' ')
  {
        /* pstr-10 */ /* pstr-11 */
break;
  }
  /* pstr-12 */
  }
/* pstr-9 */
s[1] = '\0';
printf("%s\n", str);
}

/*********************************************************************/
jan1(int yr)
{
register y, d;
/*
 * normal gregorian calendar
 * one extra day per four years
 */
y = yr;
d = 4+y+(y+3)/4;
/*
 * julian calendar
 * regular gregorian
 * less three days per 400
 */
if(y > 1800) {
  /* jan1-1 */
d -= (y-1701)/100;
d += (y-1601)/400;
}
/* jan1-2 */ /* jan1-3 */
/*
```

```
 * great calendar changeover instant
 */
if(y > 1752)
  {
    /* jan1-4 */
d += 3;
  }
/* jan1-5 */ /* jan1-6 */

return(d%7);
```

C.3.2 Função *jan1*

```
------------ Arquivo cb-arcs.tes  --------------------

1 (1,2) (y > 1800)
2 (1,3) (y < 1800)
3 (1,3) (y == 1800)
4 (3,4) (y > 1752)
5 (3,5) (y < 1752)
6 (3,5) (Y == 1752)



------------ Arquivo cb-puassoc.tes --------------------

ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS CB-POT-USOS E CB-POT-USOS/DU



Associacoes requeridas pelo Grafo( 1)

 1-1)  <1,(1,3),{ yr, y, d, dayw, smon, mon }>
 1-2)  <1,(1,3),{ yr, y, d, dayw, smon, mon }> (y < 1800)
 1-3)  <1,(1,3),{ yr, y, d, dayw, smon, mon }> (y == 1800)
 2-1)  <1,(3,5),{ yr, y, d, dayw, smon, mon }>
 2-2)  <1,(3,5),{ yr, y, d, dayw, smon, mon }> (y < 1752)
 3-1)  <1,(3,5),{ yr, y, dayw, smon, mon }>
 3-2)  <1,(3,5),{ yr, y, dayw, smon, mon }> (y < 1752)
 4-1)  <1,(3,4),{ yr, y, d, dayw, smon, mon }>
 4-2)  <1,(3,4),{ yr, y, d, dayw, smon, mon }> (y > 1752)
```

```
 5-1)  <1,(3,4),{ yr, y, dayw, smon, mon }>
 5-2)  <1,(3,4),{ yr, y, dayw, smon, mon }> (y > 1752)
 6-1)  <1,(1,2),{ yr, y, d, dayw, smon, mon }>
 6-2)  <1,(1,2),{ yr, y, d, dayw, smon, mon }> (y > 1800)


Associacoes requeridas pelo Grafo( 2)

 7-1)  <2,(3,5),{ d }>
 7-2)  <2,(3,5),{ d }> (y < 1752)
 8-1)  <2,(3,4),{ d }>
 8-2)  <2,(3,4),{ d }> (y > 1752)


Associacoes requeridas pelo Grafo( 4)

 9-1)  <4,( , ),{ d }>


------------ Arquivo cb-puexe.tes --------------------

          CB-ASSOCIACOES EXECUTADAS
 1-1)  <1,(1,3),{ yr, y, d, dayw, smon, mon }>
 1-2)  <1,(1,3),{ yr, y, d, dayw, smon, mon }> (y < 1800)
 1-3)  <1,(1,3),{ yr, y, d, dayw, smon, mon }> (y == 1800)
 2-1)  <1,(3,5),{ yr, y, d, dayw, smon, mon }>
 2-2)  <1,(3,5),{ yr, y, d, dayw, smon, mon }> (y < 1752)
 3-1)  <1,(3,5),{ yr, y, dayw, smon, mon }>
 3-2)  <1,(3,5),{ yr, y, dayw, smon, mon }> (y < 1752)
 4-1)  <1,(3,4),{ yr, y, d, dayw, smon, mon }>
 4-2)  <1,(3,4),{ yr, y, d, dayw, smon, mon }> (y > 1752)
 5-1)  <1,(3,4),{ yr, y, dayw, smon, mon }>
 5-2)  <1,(3,4),{ yr, y, dayw, smon, mon }> (y > 1752)
 6-1)  <1,(1,2),{ yr, y, d, dayw, smon, mon }>
 6-2)  <1,(1,2),{ yr, y, d, dayw, smon, mon }> (y > 1800)
 8-1)  <2,(3,4),{ d }>
 8-2)  <2,(3,4),{ d }> (y > 1752)
 9-1)  <4,( , ),{ d }>
***       Cobertura Total = 88.888889


------------ Arquivo cb-puoutput.tes -----------------

  CB-ASSOCIACOES NAO EXECUTADAS
```

```
 7-1)  <2,(3,5),{ d }>
 7-2)  <2,(3,5),{ d }> (y < 1752)
***       Cobertura Total = 88.888889
```

C.3.3 Função *pstr*

```
------------ Arquivo cb-arcs.tes --------------------

1 (2,3) (i+1 > 0)
2 (2,3) (i+1 < 0)
3 (2,6) (i+1 == 0)
4 (3,4) (*(s-1) == '\0')
5 (3,5) (*(s-1) < '\0')
6 (3,5) (*(s-1) > '\0')
7 (7,8) (i+1 > 0)
8 (7,8) (i+1 < 0)
9 (7,11) (i+1 == 0)
10 (8,9) (*s > ' ')
11 (8,9) (*s < ' ')
12 (8,10) (*s == ' ')

------------ Arquivo cb-puassoc.tes --------------------

ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS CB-POT-USOS E CB-POT-USOS/DU



Associacoes requeridas pelo Grafo( 1)

 1-1)  <1,(7,11),{ str, n, s, dayw, smon, mon }>
 1-2)  <1,(7,11),{ str, n, s, dayw, smon, mon }> (i+1 == 0)
 2-1)  <1,(10,7),{ str, n, s, dayw, smon, mon }>
 3-1)  <1,(8,10),{ str, n, s, dayw, smon, mon }>
 4-1)  <1,(8,9),{ str, n, s, dayw, smon, mon }>
 4-2)  <1,(8,9),{ str, n, s, dayw, smon, mon }> (*s > ' ')
 4-3)  <1,(8,9),{ str, n, s, dayw, smon, mon }> (*s < ' ')
 5-1)  <1,(3,5),{ str, n, s, dayw, smon, mon }>
 5-2)  <1,(3,5),{ str, n, s, dayw, smon, mon }> (*(s-1) < '\0')
 5-3)  <1,(3,5),{ str, n, s, dayw, smon, mon }> (*(s-1) > '\0')
 6-1)  <1,(5,2),{ str, n, s, dayw, smon, mon }>
```

```
 7-1)  <1,(3,4),{ str, n, s, dayw, smon, mon }>
 7-2)  <1,(3,4),{ str, n, s, dayw, smon, mon }> (*(s-1) == '\0')
 8-1)  <1,(1,2),{ str, n, i, s, dayw, smon, mon }>


Associacoes requeridas pelo Grafo( 2)

 9-1)  <2,(2,6),{ i }>
 9-2)  <2,(2,6),{ i }> (i+1 == 0)
 10-1)  <2,(3,5),{ i }>
 10-2)  <2,(3,5),{ i }> (*(s-1) < '\0')
 10-3)  <2,(3,5),{ i }> (*(s-1) > '\0')
 11-1)  <2,(5,2),{ i }>
 12-1)  <2,(3,4),{ i }>
 12-2)  <2,(3,4),{ i }> (*(s-1) == '\0')


Associacoes requeridas pelo Grafo( 6)

 13-1)  <6,(6,7),{ i }>


Associacoes requeridas pelo Grafo( 7)

 14-1)  <7,(7,11),{ i }>
 14-2)  <7,(7,11),{ i }> (i+1 == 0)
 15-1)  <7,(10,7),{ i }>
 16-1)  <7,(8,10),{ i }>
 17-1)  <7,(8,9),{ i }>
 17-2)  <7,(8,9),{ i }> (*s > ' ')
 17-3)  <7,(8,9),{ i }> (*s < ' ')


---------- Arquivo cb-puexe.tes ----------


         CB-ASSOCIACOES EXECUTADAS
 1-1)  <1,(7,11),{ str, n, s, dayw, smon, mon }>
 1-2)  <1,(7,11),{ str, n, s, dayw, smon, mon }> (i+1 == 0)
 10-1)  <2,(3,5),{ i }>
 10-3)  <2,(3,5),{ i }> (*(s-1) > '\0')
 11-1)  <2,(5,2),{ i }>
 12-1)  <2,(3,4),{ i }>
 12-2)  <2,(3,4),{ i }> (*(s-1) == '\0')
```

```
 13-1)  <6,(6,7),{ i }>
 14-1)  <7,(7,11),{ i }>
 14-2)  <7,(7,11),{ i }> (i+1 == 0)
 15-1)  <7,(10,7),{ i }>
 16-1)  <7,(8,10),{ i }>
 17-1)  <7,(8,9),{ i }>
 17-2)  <7,(8,9),{ i }> (*s > ' ')
  2-1)  <1,(10,7),{ str, n, s, dayw, smon, mon }>
  3-1)  <1,(8,10),{ str, n, s, dayw, smon, mon }>
  4-1)  <1,(8,9),{ str, n, s, dayw, smon, mon }>
  4-2)  <1,(8,9),{ str, n, s, dayw, smon, mon }> (*s > ' ')
  5-1)  <1,(3,5),{ str, n, s, dayw, smon, mon }>
  5-3)  <1,(3,5),{ str, n, s, dayw, smon, mon }> (*(s-1) > '\0')
  6-1)  <1,(5,2),{ str, n, s, dayw, smon, mon }>
  7-1)  <1,(3,4),{ str, n, s, dayw, smon, mon }>
  7-2)  <1,(3,4),{ str, n, s, dayw, smon, mon }> (*(s-1) == '\0')
  8-1)  <1,(1,2),{ str, n, i, s, dayw, smon, mon }>
  9-1)  <2,(2,6),{ i }>
  9-2)  <2,(2,6),{ i }> (i+1 == 0)
***       Cobertura Total = 86.666667

------------ Arquivo cb-puoutput.tes --------------------

 CB-ASSOCIACOES NAO EXECUTADAS

 10-2)  <2,(3,5),{ i }> (*(s-1) < '\0')
 17-3)  <7,(8,9),{ i }> (*s < ' ')
  4-3)  <1,(8,9),{ str, n, s, dayw, smon, mon }> (*s < ' ')
  5-2)  <1,(3,5),{ str, n, s, dayw, smon, mon }> (*(s-1) < '\0')
***       Cobertura Total = 86.666667
```