Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Estratégias de Seleção de Caminhos no Contexto de Critérios Estruturais de Teste de Software

Letícia Mara Peres

Profa. Dra. Silvia Regina Vergilio (Orientadora)
Departamento de Informática - UFPR

Prof. Dr. Mario Jino (Co-orientador)
DCA - FEEC - Unicamp

Campinas - São Paulo - Brasil
1999

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Estratégias de Seleção de Caminhos no Contexto de Critérios Estruturais de Teste de Software

Letícia Mara Peres[1]

Profa. Dra. Silvia Regina Vergilio (Orientadora)
Departamento de Informática - UFPR

Prof. Dr. Mario Jino (Co-orientador)
DCA - FEEC - Unicamp

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação (FEEC) da Universidade Estadual de Campinas (Unicamp), como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica e Computação.

Campinas - São Paulo - Brasil
1999

[1]Financiado parcialmente pelo CNPq e ITAI - Foz do Iguaçu/PR

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

P415e

Peres, Letícia Mara
Estratégias de seleção de caminhos no contexto de critérios estruturais de teste de software / Letícia Mara Peres.--Campinas, SP: [s.n.], 1999.

Orientadores: Silvia Regina Vergilio, Mario Jino.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Programas de computador - Testes. 2. Programas de computador - Medição. I. Vergilio, Silvia Regina. II. Jino, Mario. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Estratégias de Seleção de Caminhos no Contexto de Critérios Estruturais de Teste de Software

Letícia Mara Peres

Dissertação de Mestrado defendida e aprovada, em 17 de Dezembro de 1999, pela Banca Examinadora constituída pelos professores:

- Profa. Dra. Silvia Regina Vergilio (Orientadora)
  Departamento de Informática - UFPR
- Prof. Dr. Fernando José von Zuben
  DCA - FEEC - Unicamp
- Prof. Dr. Márcio Luiz de Andrade Netto
  DCA - FEEC - Unicamp
- Prof. Dr. Ricardo Ribeiro Gudwin (Suplente)
  DCA - FEEC - Unicamp

Resumo

Critérios estruturais de teste têm o objetivo de auxiliar a etapa de geração de dados de teste e de avaliar a adequação de um conjunto de casos de teste, oferecendo medidas de cobertura. Eles requerem a execução de caminhos do programa que exercitem alguns elementos, tais como: comandos, decisões, definições e usos de variáveis. A seleção de caminhos e conseqüente geração de dados de teste para aplicação de um critério estrutural é uma das etapas mais difíceis de serem automatizadas. Pois, é indecidível determinar um dado de entrada para executar um particular caminho em um programa; é indecidível determinar até mesmo se esse dado existe, ou seja, se o caminho é ou não executável. Isto, aliado à eficácia dos dados gerados, aumenta a importância dessa etapa e consequentemente os custos de teste. Por isso, vários trabalhos na literatura ressaltam a importância de estratégias para minimizar o número de caminhos não executáveis selecionados para satisfazer um dado critério estrutural.

Este trabalho tem como objetivos estudar, propor e fornecer mecanismos para automatização e validação de estratégias de seleção de caminhos a serem utilizadas em conjunto com critérios de teste estrutural. São propostas estratégias que consideram diferentes características de programas para seleção de caminhos visando: aumentar a eficácia, facilitar a etapa de geração de dados, e reduzir os efeitos causados por caminhos não executáveis no teste de software. Uma estrutura de representação/automatização dessas estratégias é apresentada e um módulo que implementa essa estrutura é descrito. O módulo é uma extensão à ferramenta Poke-Tool que apóia a utilizaçao de diferentes critérios de teste e foi utilizado para avaliar uma estratégia proposta com o objetivo de minimizar o número de caminhos não executáveis selecionados. Esta aplicação permitiu detectar algumas estruturas de programa para os quais a estratégia avaliada não alcança seu objetivo e também verificar aspectos de eficácia dos dados gerados para a execução dos caminhos selecionados.

Abstract

Testing criteria are useful in the task of test case generation and they are predicates to consider the testing activity ended, that is, to determine the adequation of a test set. They require the execution of paths in the program that exercise some elements such as statements, decisions, definitions and uses of variables.

Selecting paths and generating automatic test data for a given structural criterion are very hard activities since it is not always possible to determine a data that executes a particular path in a program; it is even undecidable whether this data exists, that is, whether the path is feasible. This makes those activities difficult and increases the cost of testing. Several researches attach importance to strategies that minimize the number of infeasible selected paths to cover structural criteria.

The goal of this work is to study, propose and offer strategies to select paths and mechanisms to automate and validate these strategies to be used with structural criteria. The proposed strategies consider different characteristics of programs for paths selection with the goal of: increasing the efficacy, easing the test data generation and reducing the effects caused by infeasible paths in the software testing. A structure to represent these strategies are presented. A module that implements this structure is described. This module is an extension to the testing tool named Poke-Tool, that supports different structural testing criteria. It was used to evaluate a strategy that proposes to minimize the number of infeasible selected paths. This application was guided to the detection of some program structures for which the evaluated strategy did not meet its objective and pointed out some results about the efficacy of the generated test data.

Agradecimentos

À Silvia, pela compreensão, dedicação e por tornar este trabalho possível.

Ao Jino, pelo carinho e apoio, apesar da distância. Ao Maldonado, pelas sugestões e opiniões.

Ao CNPq e ITAI, pelo suporte financeiro.

Aos colegas e amigos adquiridos na Unicamp: Paulo Bueno, pela amizade e apoio burocrático; colegas do grupo de teste de software: Adalberto, Chaim, Cristina, Cristina, Dino, Inês, Ivan, Nelson, Plinio; Antonio do LCA; Rosana do Dino, Lucimara, Miriam e Morte; Adela e Tutumi; e muitos outros.

Aos professores do Departamento de Informática da UFPR pelo carinho e incentivo. Em especial ainda ao Marcos, à Olga e ao Alexandre D. por me permitirem usar os laboratórios.

Aos colegas professores e funcionários da Unioeste e do ITAI. Aos amigos Habib e Juan, Shiro, Huei, João, Fernando, Gil, José Luiz e os outros professores do departamento, bem como a Neide e a Angelita, todos dispostos a nos ajudar sempre. Tem ainda o Dr. Woo, pela amizade.

Às minhas familias:

- Tambascia, que sempre nos acolheu com muito amor e carinho. À Cláudia, querida irmã, Lu, Rafa, D. Wanda e "Seo" Cláudio (torcemos para você continuar se superando sempre).
- Dinnouti: Léo e Gisane, nossos irmãos eternamente. Pelo apoio do Léo no Lex, Yacc e Latex e o auxílio da Gisane no inglês. Que não nos afastemos muito...
- Silva: pelo entendimento, torcida e carinho. Silvia e Luir, Olga e Luciano, Fernando. Aos tios e primos maravilhosos que ganhei desta família. Em especial ao Luciano, que arranjou um tempo e me ajudou num módulo de apoio de comparação.
- Anjos e Peres Molino: Vô Teotímio e meus tios e primos, em especial à tia Dauva. E aos distantes, meu carinho e constante lembrança...
- Peres: agradeço por ter nascido entre vocês, maravilhosos e espetaculares. Aos meus pais Ayda e Cesário, que sempre entenderam a ausência e sempre me apoiaram, me deram amor, compreensão, educação e todos os valores que hoje considero mais importantes na minha vida. Aos meus irmãos Aida, Jefferson, Ewerton e Emerson: o amor de vocês foi fundamental para a continuidade deste trabalho, e também de seus companheiros Fabiano, Ivane, Edilza e Jandi. Não posso me esquecer da alegria que seus filhos Alexandre, Thiago, Emmanuel, Lucas, Tiago, Livia e Fabrício me trouxeram.

Ao Fabiano, que mudou a minha vida.
Por todos os momentos de crescimento que me proporciona.
Com amor, carinho e amizade.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A Engenharia de Software tem se caracterizado nas últimas décadas pela busca incessante de métodos, procedimentos e ferramentas para a produção de software, provocada pela crescente demanda por qualidade e produtividade. O processo de desenvolvimento de software envolve uma série de atividades, estando sujeito a uma série de tipos de erros. Atividades de garantia de qualidade de software têm sido introduzidas ao longo do processo de desenvolvimento, entre elas a atividade de teste de software.

O teste de software tem como objetivo principal revelar a presença de erros ou defeitos no produto [38], podendo fornecer informações para as atividades de depuração e manutenção e evidências da confiabilidade do software em complemento a outras atividades. Esta atividade é considerada ainda um elemento essencial para a ascenção ao nível 3 do Modelo CMM do SEI [10]. No entanto, o teste de software é uma das atividades mais onerosas da produção de software [38, 45].

Idealmente o programa deveria ser testado com todos os valores de entrada possíveis, mas o teste exaustivo é impraticável, devido a restrições de tempo e custo [28]. Por isso, a aplicação planejada e sistemática de técnicas, critérios e ferramentas torna-se fundamental na tentativa de reduzir os custos associados à atividade de teste.

Técnicas têm sido propostas para determinar o subconjunto dos possíveis casos de teste que possa detectar o maior número de erros, com um mínimo de tempo e esforço. Tais técnicas diferenciam-se pela origem da informação utilizada na avaliação e construção de conjuntos de

casos de teste: a técnica de teste estrutural (ou teste de caixa branca) deriva os casos de teste a partir da estrutura interna dos programas; a técnica de teste funcional (ou teste de caixa preta) deriva os casos de teste a partir de requisitos funcionais do programa, sem o conhecimento da estrutura interna do programa; e a técnica baseada em erros deriva os casos de teste a partir das informações sobre os tipos de erros mais freqüentes no processo de desenvolvimento de software.

Detecta-se, então, que existem duas questões relacionadas à fase de teste: como selecionar os dados de teste e como saber se um programa foi testado o suficiente [48]. À primeira questão está associado um método M para escolher casos de teste e à segunda um critério C de adequação de um dado conjunto de casos de teste, que é um predicado que deve ser satisfeito para se considerar um programa testado o suficiente. Frankl [16] observa que existe uma correspondência entre um método de seleção de dados de testes e um critério de adequação. Dado um critério de adequação C, existe um método M_c que diz: "selecione um conjunto de teste que satisfaz C" e dado um método M, existe um critério C_m que diz: "um conjunto de teste é adequado se ele foi gerado pelo método M". Nesta dissertação o termo *critério* será usado com ambos os sentidos.

Critérios estruturais de teste estão baseados no conhecimento da estrutura interna da implementação. O objetivo é caracterizar um conjunto de componentes de um programa (elementos requeridos) que devem ser exercitados pelo conjunto de casos de teste, para se considerar o critério satisfeito.

Os critérios estruturais mais utilizados podem ser classificados em: critérios baseados em análise de fluxo de controle [47] e critérios baseados em análise de fluxo de dados [47, 48, 22, 26, 41, 28]. Critérios baseados em fluxo de dados são derivados em função de pontos onde variáveis são definidas até usos (ou possíveis usos) destas definições, ou seja, caminhos conectando estes pontos devem ser executados de modo a exercitar os elementos requeridos por estes critérios. Na análise do fluxo de dados estudam-se as maneiras pelas quais os valores atribuídos a cada variável podem afetar a execução do programa e foi utilizada originalmente na otimização de código por compiladores. Como alguns critérios baseados em fluxo de dados, podem ser citados os da Família de Critérios de Fluxo de Dados, proposta por Rapps e Weyuker [47, 48] e os da

Família Potenciais Usos, proposta por Maldonado, Jino e Chaim [29, 28].

Na prática, a satisfação completa dos critérios de teste é dificultada pois o conjunto de elementos exigidos pelos critérios estruturais em geral contém elementos não executáveis. Um elemento é não executável se não existe um caminho executável que o cubra; um caminho é executável se existem dados de entrada que causem a sua execução, caso contrário ele é dito ser não executável [16]. Não existe algoritmo para a detecção de executabilidade de um determinado caminho, pois esta é uma questão indecidível [17]. Em experimentos encontrados na literatura [28, 54, 59, 58] tem-se detectado aproximadamente 55% de caminhos não executáveis entre o total de caminhos nas rotinas testadas. Estes trabalhos relatam o alto esforço e o grande tempo gastos em gerar e identificar manualmente a executabilidade de caminhos.

Malevris *et al.* [32], objetivando prever não executabilidade, estudam a relação entre o número de predicados de um caminho e sua executabilidade em um conjunto de rotinas e evidenciam a validade de sua hipótese: quanto maior o número de predicados de um caminho, maior a probabilidade de ele ser não executável. O conjunto de caminhos analisado tinha o objetivo de satisfazer o critério estrutural todos LCSAJ's [17], baseado no fluxo de controle do programa. Vergilio *et al.* [53] realizaram um estudo análogo e concluíram a validade da mesma hipótese no contexto dos Critérios Potenciais Usos.

A atividade de geração de dados de teste para satisfazer um critério estrutural é uma das etapas mais difíceis de ser automatizada. Uma questão importante que afeta diretamente a geração de dados de teste baseada na estrutura do código é a escolha dos caminhos que cobrem os elementos requeridos. Isto porque não é possível gerar dados de teste para um caminho não executável e não se pode garantir a capacidade de um dado critério em revelar defeitos. A escolha de um determinado caminho pode influenciar no tempo gasto para um critério ser satisfeito e na eficácia do critério, ou seja, o número de erros revelados durante o teste [55].

O desenvolvimento de ferramentas de auxílio à aplicação de critérios de teste são indispensáveis ao aumento da qualidade da produção do software. Várias ferramentas de teste têm sido desenvolvidas para o teste estrutural: Proteste [46], Atac [23], Asset [16] e Poke-Tool [11]. Esta última ferramenta dá apoio à utilização dos Critérios Potenciais Usos, critérios baseados

em fluxo de dados, além de outros critérios baseados em fluxo de controle. Uma versão comercial de ferramenta para teste estrutural está disponível, a XSuds [1]. Algumas das ferramentas de teste citadas, tais como Asset e Poke-Tool, implementam heurísticas de caracterização de não executabilidade de elementos [54]; porém, nenhuma das ferramentas de teste estrutural citadas implementam a geração automática de caminhos completos que cubram elementos requeridos por um dado critério como um meio de facilitar a geração de dados de teste.

A geração automática de caminhos deve se utilizar de estratégias de seleção como um meio de escolher caminhos completos a partir de um conjunto de caminhos que cobrem um determinado elemento requerido por um dado critério. Yates e Malevris [62] propuseram uma estratégia para satisfazer o critério todos os ramos, selecionando um conjunto de caminhos com maior probabilidade de serem executáveis. A idéia é selecionar, entre dois caminhos candidatos a cobrir um elemento requerido por um critério, o de menor número de predicados, pois esse tem maior probabilidade de ser executável.

Exemplos de outras estratégias de seleção de caminhos são: a estratégia aleatória, que apesar de não auxiliar a determinação de caminhos não executáveis, é menos custosa, mais prática e mais fácil de automatizar [36, 6]; a estratégia de minimização de casos de teste, proposta por Bertolino e Marré [3, 4], que propõem um algoritmo de geração de caminhos mas não analisam a eficácia de sua estratégia.

1.1 Motivação

Pode-se destacar, da discussão anterior, alguns fatores que serviram como motivação para este trabalho:

- erros estão presentes na maioria dos programas e está clara a necessidade de automatização de técnicas/critérios de teste para obtenção de produtos de qualidade e de baixo custo;
- determinar caminhos completos que cubram elementos requeridos por um dado critério estrutural é uma atividade necessária para a geração automática de dados de teste;
- caminhos não executáveis estão presentes na maioria dos programas, até mesmo os bem

formulados, e é importante a aplicação de estratégias que reduzam seus efeitos;

- gerar caminhos que cubram elementos requeridos por critérios estruturais e identificar não executabilidade manualmente são atividades demoradas e que consomem bastante esforço e tempo;
- a inexistência de ferramentas para critérios baseados em fluxo de dados que automatizem a geração de caminhos completos ou reduzam a geração de caminhos não executáveis;
- escolher uma estratégia de seleção de caminhos a ser usada com um critério de teste estrutural é uma questão essencial para facilitar a atividade de teste e/ou aumentar a eficácia dos dados de teste gerados;
- necessidade de fornecer mecanismos para automatizar/validar possíveis estratégias de seleção de caminhos, entre elas a estratégia menor número de predicados proposta por Yates e Malevris [62].

1.2 Objetivos

O objetivo do trabalho é estudar, propor e fornecer mecanismos para automatização e validação de estratégias de seleção de caminhos a serem utilizadas com critérios de teste estrutural, de modo a diminuir o esforço de teste, inclusive com redução do número de caminhos não executáveis. Isso compreende:

- estudo de estratégias de seleção de caminhos e suas características, existentes na literatura;
- automatização da estratégia menor número de predicados, de Yates e Malevris [62, 32];
- criação de um módulo de geração de caminhos completos, que implemente as estratégias de seleção de caminhos apresentadas. Esse módulo deve ser uma extensão da ferramenta Poke-Tool e deve gerar caminhos para satisfazer os critérios por ela implementados;
- validação do módulo construído, aplicando a estratégia menor número de predicados de Yates e Malevris [62, 32], observando a redução de caminhos não executáveis.

1.3 Organização

Neste capítulo foi apresentado o contexto em que este trabalho se insere, destacando-se os problemas referentes às estratégias de seleção e à geração de caminhos, motivações e objetivos desta dissertação.

No Capítulo 2, é apresentada uma revisão bibliográfica dos trabalhos relacionados, a terminologia e os conceitos básicos utilizados nesse trabalho. São descritos os principais aspectos da seleção de caminhos. Os critérios estruturais de teste são descritos em detalhe, mais particularmente os critérios baseados em fluxo de controle e os critérios Potenciais Usos que são utilizados como base nesta dissertação para a geração automática de caminhos.

No Capítulo 3, é proposta uma estrutura genérica para representar estratégias de seleção de caminhos com o objetivo de facilitar a automatização. Diferentes estratégias de seleção a serem utilizadas com critérios estruturais são descritas utilizando essa estrutura. Algumas dessas estratégias são extraídas da literatura e mencionadas no Capítulo 2. Outras estratégias estão sendo propostas nessa dissertação e mostram como diversas características, inclusive métricas de software, podem ser utilizadas para a seleção de caminhos.

No Capítulo 4, é apresentado um módulo de extensão à Poke-Tool, o Poke-Paths, que gera automaticamente caminhos completos utilizando uma dada estratégia de seleção de caminhos e implementa a estrutura de representação proposta no Capítulo 3.

No Capítulo 5, é detalhada uma aplicação do módulo Poke-Paths realizada com o objetivo de validar sua implementação e verificar a aplicação da estratégia menor número de predicados, proposta por Yates e Malevris [62, 32] com relação ao número de caminhos não executáveis. Neste capítulo é apresentada ainda uma metodologia para um experimento que avalie a eficácia dos dados de teste gerados utilizando a estratégia menor número de predicados.

No Capítulo 6, são apresentadas as conclusões e propostas de trabalhos futuros.

Este texto contém quatro apêndices referentes ao módulo implementado e suas aplicações neste trabalho. O Apêndice A apresenta alguns aspectos de utilização do módulo Poke-Paths. Os demais apêndices apresentam dados referentes à aplicação do módulo detalhada no Capítulo

5: no Apêndice B estão apresentados os códigos e medidas de complexidade de programas utilizados na aplicação realizada observando a estratégia menor número de predicados, de Yates e Malevris [62, 32] quanto ao número de caminhos não executáveis; no Apêndice C estão organizados arquivos e tabelas, de maneira mais detalhada, referentes aos resultados alcançados nesta aplicação; no Apêndice D são apresentados detalhes do programa utilizado na aplicação da metodologia de um experimento referente à eficácia, sugerida no Capítulo 5.

Capítulo 2

Revisão Bibliográfica e Conceitos Básicos

Neste capítulo são apresentados terminologia, conceitos básicos e uma revisão bibliográfica dos principais trabalhos existentes na literatura, relacionados ao teste estrutural. O objetivo deste capítulo é introduzir o conhecimento e notações que serão utilizados nesta dissertação. São enfocados critérios de teste baseados em fluxo de controle e baseados em fluxo de dados, o problema da não executabilidade e da geração automática de caminhos e ferramentas de auxílio ao teste de software.

2.1 Conceitos Básicos - Terminologia

Como apresentado por Maldonado [28], um programa P pode ser decomposto em um conjunto de **blocos** disjuntos de comandos; a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos, na ordem dada, desse bloco. Todos os comandos de um bloco, possivelmente com a exceção do primeiro, têm um único predecessor, e exatamente um único sucessor, exceto possivelmente o último comando.

A representação de um programa P como um **grafo de fluxo de controle (GFC)**, ou grafo de programa, consiste em estabelecer uma correspondência entre nós e blocos e em indicar possíveis fluxos de controle entre blocos através dos arcos. Desta forma, para o grafo $G = (N, A, e)$, N é o conjunto de blocos de comandos, a partir de agora chamados **nós**; A é o conjunto de arcos que representam o fluxo de controle do programa, a partir de agora chamados

arcos ou **ramos**; e é o nó de entrada. Considera-se que o grafo G contém um único nó de entrada e e um único nó de saída s.

Define-se um **caminho** como sendo uma seqüência finita de nós $(n_i, ..., n_k), k \geq 2$, tal que (n_i, n_{i+1}) é um arco do grafo G, para $i = 1, 2, ..., k-1$. Diz-se que um caminho é: **completo** se o primeiro nó do caminho é o nó de entrada e de G e o último nó é o nó de saída s de G; **simples** se todos os nós, exceto possivelmente o primeiro e o último, forem distintos; e **livre de laços** se todos os nós são distintos.

Define-se **predicado** como a menor unidade boleana de uma expressão.

Uma ocorrência de um identificador representando uma variável num programa pode ser de três tipos: definição, indefinição, uso.

Uma **definição** de variável ocorre se um valor é armazenado em uma posição de memória. No caso de um programa, a definição ocorre se a variável é referenciada do lado esquerdo de um comando de atribuição, ou em um comando de entrada, ou em chamadas de procedimento como parâmetro de saída (nesse caso, se a variável foi passada por referência ou por nome). Uma variável está **indefinida** quando ou seu valor se torna inacessível, ou sua localização deixa de estar definida na memória. Um **uso** de uma variável ocorre quando ela é referenciada sem ser definida e pode ser de dois tipos: c-uso(*computation-use*) e p-uso(*predicate-use*). O primeiro afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição seja observado. O segundo afeta diretamente o fluxo de controle do programa.

Seja x uma variável, um caminho $(i, n_1, ..., n_m, j)$, $m \geq 0$, é **um caminho livre de definição** com respeito a (c.r.a) x do nó i para o nó j ou do nó i para o arco (n_m, j), se i possui uma definição de x e nenhuma definição de x ocorre nos nós $n_1, ..., n_m$. O arco (i, j), para $m = 0$, é considerado um caminho livre de definição c.r.a x do nó i para o arco (i, j).

Define-se: **defg**(i) = {variáveis v | v é definida no nó i}; **pdcu**(x,i) = {nós j | existe um caminho livre de definição c.r.a x do nó i para o nó j}; **pdpu**(x,i) = {arcos (j,k) | existe um caminho livre de definição c.r.a x de i para o arco (j,k)}; **potencial du-caminho** c.r.a x como um caminho livre de definição $(n_1, n_2, ..., n_j, n_k)$ c.r.a x do nó n_1 para o nó n_k e para o arco (n_j, n_k), onde o caminho $(n_1, n_2, ..., n_j)$ é um caminho livre de laços e no nó n_1

ocorre uma definição de x; **associação potencial-definição-c-uso** como a tripla $[i, j, x]$ onde $x \in defg(i)$ e $j \in pdcu(x, i)$; **associação potencial-definição-p-uso** como a tripla $[i, (j, k), x]$ onde $x \in defg(i)$ e $(j, k) \in pdpu(x, i)$; e **potencial associação** como uma associação potencial-definição-c-uso, uma potencial-associação-p-uso ou um potencial-du-caminho [28].

A extensão do grafo de fluxo de controle G com informações de fluxo de dados (tipos de ocorrências de variáveis) consiste essencialmente em associar a cada nó i de G o conjunto $defg(i)$; o grafo estendido com informações de ocorrências de definição de variáveis é denominado **grafo def**, aqui também chamado de G.

Maldonado [28] introduziu a notação $[i, (j, k), \{v_1, ...v_n\}]$ para representar o conjunto de associações $[i, (j, k), v_1]$,...,$[i, (j, k), v_n]$; ou seja, $[i, (j, k), \{v_1, ..., v_n\}]$ indica que existe no grafo G, pelo menos um caminho livre de definição c.r.a $v_1, ..., v_n$ do nó i ao arco (j, k). Observe-se que podem existir em G outros caminhos livres de definição c.r.a algumas das variáveis $v_1, ..., v_n$ mas que não sejam, simultaneamente livres de definição para todas as variáveis $v_1, ..., v_n$.

Diz-se que um caminho $\pi_1 = (i_1, ..., i_k)$ está incluído em um conjunto Π de caminhos se e somente se Π contém um caminho $\pi_2 = (n_1, ..., n_m)$ tal que $i_1 = n_j, i_2 = n_{j+1}, ..., i_k = n_{j+k-1}$, para algum j, $1 \leq j \leq m-k+1$. Diz-se que π_1 **está incluído** em π_2 ou que π_1 é um **subcaminho** de π_2.

Um caminho completo π **cobre** uma associação potencial-definição-c-uso $[i, k, x]$, (respectivamente, uma associação potencial-definição-p-uso $[i, (j, k), x]$) se ele incluir um caminho livre de definição c.r.a x do nó i para o nó k (respectivamente do nó i para o arco (j, k)). O caminho π cobre um potencial-du-caminho π_1 se π_1 estiver incluído em π. Um conjunto Π de caminhos cobre uma potencial-associação se algum elemento do conjunto o fizer. Note-se que, se um conjunto de caminhos Π cobrir uma potencial-associação do nó i para o nó k (respectivamente do nó i para o arco(j, k)), então existe um caminho livre de definição c.r.a variável $x \in defg(i)$ do nó i para o nó k (respectivamente, para o arco (j, k)), que está incluído em Π.

Um caminho completo é **executável** ou **factível** se existe um conjunto de valores que possa ser atribuído às variáveis de entrada, variáveis globais e parâmetros do programa, que cause a execução desse caminho; caso contrário, diz-se que ele é **não executável** [16]. Um caminho é

executável se ele for um subcaminho de um caminho completo executável, isto é, se ele estiver incluído em um caminho completo executável. Uma potencial associação é executável se existir um caminho completo executável que cubra essa associação; caso contrário é não executável. Em função desses conceitos outros conjuntos são definidos: **fpdcu(x,i)** = $\{j \in$ pdcu(x,i) | a potencial-associação-c-uso (i, j, x) é executável$\}$; **fpdpu(x,i)** = $\{(j, k) \in$ pdpu(x,i) | a potencial-associação-p-uso $(i, (j, k), x)$ é executável$\}$.

Sejam $\pi_1 = (i, n_1, ..., n_m, j)$ e $\pi_2 = (j, m_1, ..., m_m, l)$ definidos em G, a **concatenação** de π_1 e π_2, denotada por $\pi_1 \cdot \pi_2$, é definida como sendo o caminho $(i, n_1, ..., n_m, j, m_1, ..., m_m, l)$ em G.

2.2 Critérios Estruturais

Um critério de seleção de casos de teste é tal que pode ser seguido para selecionar um subconjunto finito e factível de casos de teste de um domínio de execução potencialmente infinito [3]. Um critério de adequação é um predicado que deve ser satisfeito para considerar um programa testado o suficiente [48]. Como Frankl [16] observa, existe uma correspondência entre um método de seleção de dados de testes e um critério de adequação. Deste modo, um critério C pode ser usado como critério de seleção e de adequação. No teste estrutural, um conjunto de caminhos deve ser executado de modo a exercitar elementos do programa, dependendo de um critério em particular. Nas próximas subseções são apresentados alguns dos principais critérios estruturais.

Segundo Frankl e Weyuker [17], uma das propriedades que deve ser satisfeita por um bom critério de teste é a *aplicabilidade*; diz-se que um critério C satisfaz a propriedade de aplicabilidade se para todo programa P existe um conjunto de casos de teste T que é C-adequado para P, ou seja, o conjunto Π de caminhos executados por T cobre cada elemento exigido pelo critério C. Para satisfazer esta propriedade, para cada critério C, um critério C^*, que requer apenas elementos executáveis, é definido. Os critérios C^* são referenciados como critérios executáveis e não são descritos neste texto.

2.2.1 Critérios Baseados em Fluxo de Controle

Os critérios de teste estrutural baseados em fluxo de controle utilizam unicamente informações de fluxo de controle do grafo para derivar os requisitos de teste.

Critério todos-nós - Requer que cada nó do grafo G seja executado ao menos uma vez.

Critério todos-ramos - Requer que cada ramo (arco) do grafo G seja executado ao menos uma vez.

Critério todos-caminhos - Requer a execução de todos os caminhos do grafo G.

2.2.2 Critérios Baseados em Fluxo de Dados

Os critérios baseados em análise de fluxo de dados utilizam a informação de fluxo de dados como fonte de informação para derivar os requisitos de teste e requerem que as interações que envolvem definições de variáveis de programa e subseqüentes referências a estas definições sejam testadas. Esses critérios baseiam-se portanto, para a derivação de casos de teste, nas associações entre a definição de uma variável e os seus possíveis usos subseqüentes.

Diferentes critérios baseados em fluxo de dados foram propostos por Herman [22], Laski e Korel [26], Ntafos [41], Ural e Yang [52]. Rapps e Weyuker [47, 59, 48] definiram a Família de Critérios de Fluxo de Dados (DFCF); os três critérios centrais dessa família são: todas-defs, todos-usos, todos du-caminhos. Outros três critérios todos-p-usos, todos-p-usos/alguns c-usos, todos-c-usos/alguns p-usos são variações do critério todos-usos. Maldonado, Chaim e Jino [29, 30, 28], introduziram a família de critérios Potenciais Usos, que está baseada no conceito de *potencial uso*, que diz: se uma definição de variável ocorre, existe um potencial uso a ela relacionado. Esses critérios são utilizados nessa dissertação para exemplificar e validar estratégias de seleção de caminhos e por isso são apresentados detalhadamente a seguir.

Critérios Potenciais Usos

Os critérios Potenciais Usos requerem basicamente que caminhos livres de definição, em relação a qualquer nó i que possua uma definição de variável e a qualquer variável x definida em i, sejam executados, independentemente de ocorrer uso dessa variável nesses caminhos. Neste sentido,

pode-se verificar, por exemplo, que o valor de x não foi alterado nesses caminhos (possivelmente devido a efeitos colaterais) ganhando-se, desta forma, maior confiança de que a computação correta foi realizada.

Seja Π um conjunto de caminhos completos:

Critério todos-potenciais-usos - Π satisfaz o critério todos-potenciais-usos se para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui pelo menos um caminho livre de definição c.r.a x do nó i para todo nó e para todo arco possível de ser alcançado a partir do nó i. Equivalentemente, em termos do conceito de potencial associação, Π deve cobrir todas as potenciais associações $[i, j, x] \mid j \in$ pdcu(x,i) e todas as potenciais associações $[i(j,k), x] \mid (j,k) \in$ pdpu(x,i) para cada nó $i \in G$, defg(i) $\neq \emptyset$ e para cada $x \in$ def(i).

Critério todos-potenciais-usos/du - Π satisfaz o critério todos-potenciais-usos/du se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui pelo menos um potencial-du-caminho c.r.a x do nó i para todo arco possível de ser alcançado a partir do nó i. Equivalentemente, em termos do conceito de potencial associação, Π deve incluir, para cada nó $i \in$ G $\mid$ defg(i) $\neq \emptyset$, um potencial-du-caminho de i para (j,k) c.r.a x para todas as potenciais associações $[i(j,k), x] \mid (j,k) \in$ pdpu(x,i).

Critério todos-potenciais-du-caminhos - Π satisfaz o critério todos-potenciais-du-caminhos se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui todos os potenciais-du-caminhos c.r.a x em relação ao nó i para todo nó $j \in$ pdcu(x,i) e para todo arco (j,k) $\in$ pdpu(x,i).

Também os critérios Potenciais-Usos foram modificados para satisfazer a propriedade da aplicabilidade e definidos novos critérios denominados Critérios Potenciais Usos Executáveis - todos-potenciais-usos executáveis, todos-potenciais-usos/du executáveis, todos-potenciais-du-caminhos executáveis. Então, para cada critério Potencial Uso C define-se um novo critério C*, onde a modificação consiste em selecionar as potenciais associações requeridas utilizando-se os conjuntos fpdcu(x,i) e fpdpu(x,i); apenas elementos executáveis, portanto.

Com o objetivo principal de aplicar os critérios Potenciais Usos, foi contruída no DCA/ FEEC/UNICAMP uma ferramenta de teste, a Poke-Tool [11]. Esta ferramenta é descrita

posteriormente neste capítulo.

2.3 Não Executabilidade de Caminhos

Um caminho não executável é aquele para o qual não existe dado de teste que o execute [47, 17, 28, 54]. Estes caminhos afetam direta e indiretamente as atividades de teste do tipo estrutural, provocando o aumento de custos da geração de dados de teste. Isto acontece porque, determinar se existe ou não um dado de teste que o execute é uma questão indecidível [47].

Outro aspecto afetado pela presença de caminhos não executáveis é o do encerramento das atividades de teste. Segundo Myers[38] a satisfação de um critério de teste é um dos itens necessários para se considerar a atividade de teste encerrada (critério de parada). O problema é que a maioria dos critérios exige a execução de elementos não executáveis. Um elemento é não executável se não existir um caminho executável que o cubra. Determinar quais elementos são executáveis é indecidível, portanto nem sempre é possível satisfazer um critério, nem mesmo é possível saber se existe um conjunto de dados de teste que o satisfaça.

Ainda, segundo Frankl e Weyuker [17], os caminhos não executáveis afetam algumas propriedades dos critérios. A ordem parcial de inclusão é perturbada consideravelmente, fazendo com que vários critérios deixem de incluir o critério todos-ramos, e o uso de todo resultado computacional [17, 28]. Para a satisfação dos critérios, estes foram redefinidos, criando versões aplicáveis teoricamente [17] que requerem apenas elementos executáveis, os critérios C^*. Tal solução entretanto não se reflete na prática, visto que permanece a necessidade da identificação de quais elementos requeridos são ou não executáveis.

A associação será não executável se não existirem caminhos executáveis que a cubram. Determinar se uma associação é ou não executável é indecidível pois é necessário saber quais caminhos são executáveis [54]. Uma heurística baseada em análise de fluxo de dados foi proposta por Frankl [16] e tem como objetivo determinar se uma associação é não executável. A idéia básica é eliminar do conjunto de caminhos candidatos a cobrir a associação aqueles que não são executáveis e que possuam redefinições das variáveis da associação. A associação será não executável se o conjunto de caminhos candidatos ficar vazio. Em [54] e [9] é possível encontrar

os detalhes desta heurística, bem como considerações sobre sua implementação realizada por Vergilio e Bueno.

Um estudo foi realizado em [54], sobre a caracterização e previsão de caminhos não executáveis. Uma parte deste estudo foi baseada no experimento realizado por Hedley e Hennel [21], onde Vergilio apresenta as principais características/categorias de não executabilidade encontradas em um conjunto de programas utilizado por Weyuker [58]. Vergilio [54] e Weyuker [17, 15] encontraram em seus experimentos cerca de 55% de caminhos não executáveis entre os analisados e, além disso, a maioria dos programas testados apresentaram caminhos não executáveis. Outra parte está baseada em trabalhos de Malevris *et al.* [62, 32] e diz respeito à previsão de não executabilidade. Detalhes destes trabalhos são dados a seguir.

2.3.1 Previsão de Caminhos Não Executáveis

Malevris *et al.* [32, 62], estudando a natureza dos caminhos não executáveis de um programa no contexto do critério todos-ramos, propuseram que o número de predicados envolvidos em um caminho possa ser considerado como uma métrica para prever não executabilidade, ou seja, quanto maior o número de predicados de um caminho, maior a probabilidade de ele ser não executável.

Isto é relativamente intuitivo, considerando-se que um caminho π possua $q > 0$ predicados, para que π seja executável é necessário que os predicados sejam consistentes e consistentes entre si. Se π_1 possui q predicados e π_2 possui r predicados, tal que $r > q$, π_2 tem maior probabilidade de ser não executável que π_1, ou seja, dos seus predicados não serem consistentes.

Para verificar a validade destas afirmações, Malevris *et al.* realizaram um estudo com 36 rotinas da biblioteca NAG e analisaram 583 caminhos, chegando à conclusão de que existe uma forma probabilística de relacionamento entre o número de predicados de um caminhos e sua executabilidade.

Malevris *et al.*, considerando que a executabilidade de um caminho é indecidível, encaram a métrica **número de predicados de um caminho** como uma heurística e em [62] propõem uma estratégia para desconsiderar caminhos não executáveis, com o objetivo principal de reduzir

esforço e tempo. A idéia é selecionar um conjunto de caminhos de Π, para realizar o teste de ramos, que possui caminhos envolvendo um número mínimo de predicados, pois segundo a heurística, este seria o conjunto que tem maior probabilidade de conter um número menor de caminhos não executáveis.

Vergilio et. al. [53] realizaram um trabalho com o objetivo de explorar a veracidade da hipótese de Malevris et. al. no contexto dos critérios estruturais baseados na análise de fluxo de dados, em particular para os Critérios Potenciais Usos. A aplicação dos Critérios Potenciais Usos foi apoiada pela ferramenta de teste Poke-Tool [11]. O trabalho consistiu na aplicação de um "benchmark" que teve como objetivo verificar o comportamento e a aplicação dos Critérios Potenciais Usos em programas reais e estudar a ocorrência e a influência de caminhos não executáveis. Concluiu-se, com este trabalho, que a hipótese de Malevris et. al. também é válida no contexto dos critérios estruturais baseados em fluxo de dados, para programas escritos em linguagem C. Estes resultados foram utilizados para obter um método que auxilia na geração de dados de teste para satisfazer os critérios baseados em fluxo de dados [54]. Por exemplo, a seleção dentre os caminhos candidatos a cobrir uma associação requerida por um critério baseado em fluxo de dados, aquele que tenha o menor número de predicados, poderá reduzir os efeitos causados pelos caminhos não executáveis.

2.4 Geração Automática de Caminhos de Teste

Os métodos de Yates e Malevris [62] e Bertolino e Marré [3, 4], apresentados a seguir representam o estado da arte para as atividades de seleção e geração de caminhos para satisfazer critérios estruturais de teste de software. Os métodos descritos baseiam-se na cobertura dos critérios todos-ramos e todos-usos [48]. Estes trabalhos apresentam a geração de caminhos como uma sub-atividade de auxílio à geração de dados de teste e consideram a estratégia menor número de predicados e os resultados descritos na Seção 2.3.1.

Considera-se que para a atividade de teste é necessário selecionar um conjunto Π de caminhos através de $G = (N, A, e)$ que representa o programa c, que cubra o elemento R requerido pelo critério C. Posteriormente a isto, é derivado de Π um conjunto correspondente de dados

de teste $D(\Pi)$ que irá direcionar a execução de c para cada caminho de Π. Após esta etapa, o programa c é executado com todos os elementos de $D(\Pi)$. Métodos de geração de caminhos implementam a seleção de caminhos que comporão o conjunto Π, gerando-os e refletindo o problema da não executabilidade, descrito anteriormente.

Yates e Malevris

Em [62], é apresentado um método de geração de caminhos, denominado ESPM (Extended Shortest Path Method). Este método foi proposto para a geração de um conjunto de caminhos Π^* que cobrem um determinado arco de um grafo, requerido pelo critério todos-ramos, e que contêm um número mínimo de predicados.

Para representar uma unidade de código c, o método usa um grafo $G = (N, A, e)$ já definido anteriormente. Tem-se associado com cada arco de A um tamanho de unidade. O ESPM então identifica os caminhos requeridos através de c com um conjunto de menores caminhos de e para s em G e gera o conjunto Π^* usando técnicas de teoria dos grafos. Empregando o Princípio da Otimalidade[1][34, 33] para gerar caminhos que cubram o arco (i, j), o caminho $\pi^{(k)}_{eijs}$, onde k é a ordem do caminho no conjunto Π^*, e e s são os nós iniciais e finais do grafo, pode ser expresso como:

$$\pi^{(k)}_{eijs} = \pi^{(m)}_{ei} \cdot ij \cdot \pi^{(n)}_{js} \tag{2.1}$$

onde m e n são inteiros satisfazendo $1 \leq m \leq k$, e $1 \leq n \leq (k - m + 1)$ e o operador "." denota a operação concatenação.

São apresentadas pelos autores algumas restrições para a construção dos caminhos requeridos de uma maneira eficiente e sistemática com o auxílio da equação (2.1):

1. que nenhum arco (i, j) de $\pi^{(k)}_{eijs}$ seja duplicado;

2. que os valores de n e m correspondentes a valores específicos de k possam ser determinados eficientemente;

3. que os caminhos apropriados $\pi^{(m)}_{ei}$ e $\pi^{(n)}_{js}$ possam ser gerados eficientemente;

[1]Tradução para o português de *Optimality Principle*.

O primeiro problema é solucionado descartando qualquer caminho $\pi_{ei}^{(t)}$ e $\pi_{js}^{(n)}$ que contenha o arco (i, j). O segundo problema possui vários métodos padrões disponíveis para solucioná-lo, segundo Malevris *et al.* [32]. O terceiro item é instância de problemas bem pesquisados de determinação do k-ésimo menor caminho entre dois nós de um grafo, existente em [34, 33, 14].

Trabalho de Bertolino e Marré

Em seus trabalhos [3, 4], Bertolino e Marré apresentam duas técnicas para a geração automática de caminhos de teste que cubram os elementos requeridos pelos critérios todos-ramos e todos-usos [17].

O trabalho [3] apresenta um algoritmo generalizado que encontra um conjunto de caminhos de cobertura para um dado grafo de fluxo de um programa. A análise é conduzida em um grafo $G = (N, A, e)$ e são exploradas as relações de dominância e implicação que formam as duas árvores de arcos de G. Estas relações possibilitam identificar imediatamente um subconjunto de arcos de G, chamados *arcos irrestritos*, tendo a propriedade de que um conjunto de caminhos que exercitem todos os arcos irrestritos irão cobrir também todos os arcos em G. Os caminhos são derivados um de cada vez, cada caminho cobrindo ao menos um arco irrestrito não coberto. Os mesmos princípios são utilizados em [4] para a cobertura de *duas* irrestritas. Uma *dua* é uma associação entre a definição e o uso de uma variável.

O algoritmo utilizado por Bertolino e Marré consiste em visitar recursivamente e em combinação, as árvores de dominância e de implicação. Estes trabalhos sugerem derivar um conjunto de caminhos de cobertura para satisfazer diferentes requisitos ou critérios.

As políticas sugeridas por Bertolino e Marré são Número-Mínimo-de-Caminhos e Menor-Número-de-Predicados, esta última proposta por Yates e Malevris [62].

2.5 A Ferramenta Poke-Tool

Um levantamento das ferramentas mais significativas de apoio ao teste estrutural foi realizado em [31] e foram destacadas: a ferramenta Asset [16] (*A System to Select and Evaluate Tests*), desenvolvida na Universidade de Nova Iorque, e a ferramenta Proteste [46], desenvolvida na Universidade Federal do Rio Grande do Sul, que apóiam a Família de Critérios de Fluxo de Dados,

de Rapps e Weyuker, para programas escritos em Pascal; as ferramentas Atac [23] (*Automatic Test Analysis for C*) e sua versão comercial XSuds [1], desenvolvidas na Bell Communications Research, apóiam os critérios de Rapps e Weyuker para programas escritos em C; e a ferramenta Poke-Tool [11], desenvolvida na Universidade Estadual de Campinas, no DCA/FEEC, a partir de 1991, 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 critérios potenciais-usos, através das seguintes tarefas [9]:

1. Análise de código fonte e criação de uma base de dados;
2. Geração de relatórios com informações da análise estática e identificação da estrutura de dados e controle;
3. Apoio à execução de casos de teste;
4. Análise dos resultados do teste.

A ferramenta Poke-Tool é composta por módulos interdependentes, que geram saídas que são utilizadas pelos outros módulos. Na versão para o ambiente UNIX, as tarefas citadas estão associadas a diversos módulos integrados através de *scripts shell.*

Considerando a Figura 2.1, o módulo *Pokernel* é responsável pela análise estática do código fonte; instrumentação do programa para o teste; geração dos elementos requeridos para os critérios apoiados e de informações estáticas do programa. Para tanto são utilizados o programa na Linguagem Intermediária (LI) e o grafo de fluxo de controle (GFC), gerados pelos módulos anteriores (*Li e Nli*, respectivamente). O módulo *Pokeexec* apóia a execução do programa instrumentado, direcionando para os arquivos os dados de entrada, os parâmetros fornecidos, as saídas obtidas e os caminhos executados para cada caso de teste. O programa *Pokeaval* permite a avaliação da cobertura atingida com o conjunto de casos de teste aplicado, para o critério escolhido, além de fornecer os elementos requeridos executados e não executados.

Para determinar associações e caminhos não executáveis foram incorporadas à Poke-Tool rotinas que implementam a heurística proposta em [17], além de algumas extensões e facilidades

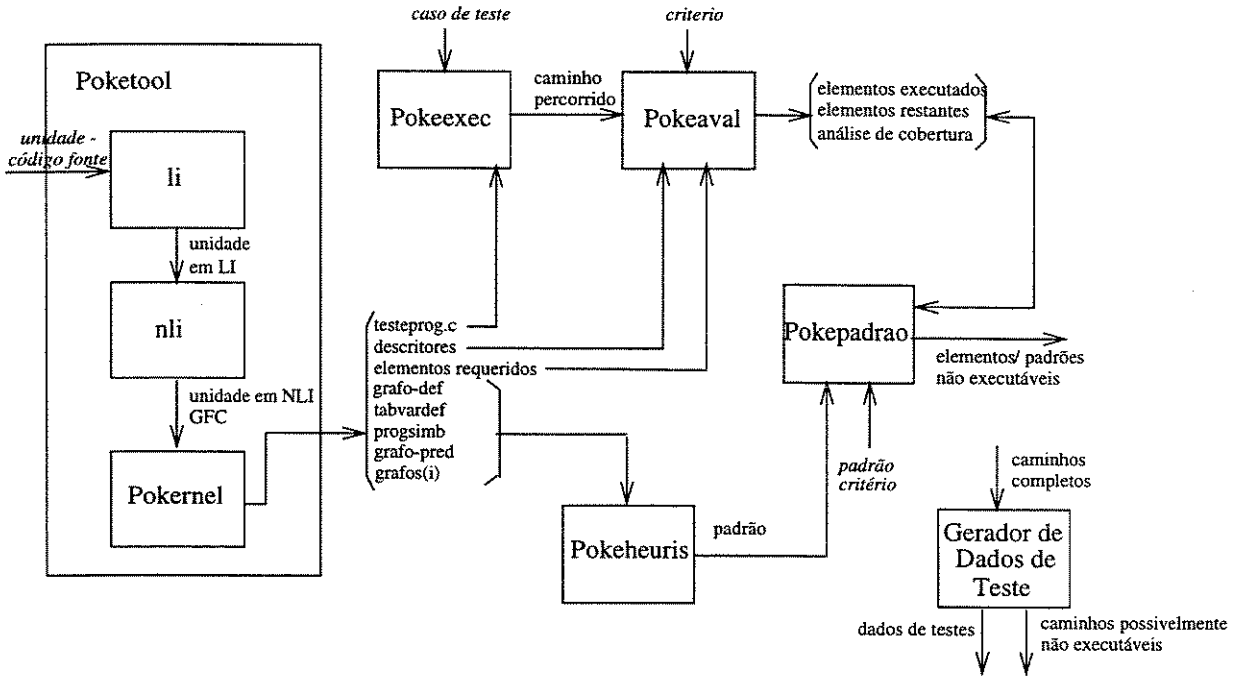

Figura 2.1: Arquitetura atual da ferramenta Poke-Tool

adicionais para o tratamento de não executabilidade. A heurística implementada pelo módulo *Pokeheuris* está baseada em técnicas de execução simbólica e análise de fluxo de dados. Para reduzir as limitações existentes na execução simbólica foi implementada uma interface com o usuário, cuja participação, em alguns casos, é fundamental. O tratamento de padrões de não executabilidade é apoiado através do módulo *Pokepadrão* [9], que retira do conjunto de elementos requeridos aqueles que são não executáveis devido a algum padrão identificado pelo testador. Estes módulos são importantes porque reduzem o trabalho do testador na identificação de elementos não executáveis, fator crítico em termos de custo [56]. Outro aspecto importante e de particular interesse, é que a eliminação *a priori* dos elementos não executáveis tende a reduzir o custo da geração automática dos dados de teste.

Um módulo de geração de dados de teste, apresentado na Figura 2.1 como "Gerador de Dados de Teste", foi desenvolvido por Bueno [7]. Este módulo utiliza algoritmos genéticos e execução dinâmica e gera dados de teste a partir de caminhos completos.

Detalhes de operação desta ferramenta podem ser encontrados no Manual do Usuário da Poke-Tool [8].

2.6 Considerações Finais

Este capítulo apresentou os principais conceitos relativos ao teste estrutural de software. Descreveu em detalhes os principais critérios da família de critérios Potenciais Usos e a ferramenta Poke-Tool que apóia a utilização dos mesmos.

A questão da não executabilidade de caminhos foi abordada, inclusive citando-se trabalhos teóricos e práticos que têm como objetivo a previsão de caminhos não executáveis. A métrica menor número de predicados sugerida por Yates e Malevris [62] pode ser utilizada como estratégia de seleção de caminhos de teste. Destaca-se o trabalho de Bertolino e Marré [3] de geração de caminhos de teste.

Os trabalhos descritos neste capítulo, em conjunto com estudos de métricas de software e teoria de grafos, serviram de base para a proposta de estratégias de seleção de caminhos. Estas estratégias têm como objetivo a seleção de caminhos para satisfazer critérios estruturais e consideram a não executabilidade de caminhos e outras características do software para a seleção de caminhos que tenham maior probabilidade em aumentar o desempenho e a eficácia do teste de software, e serão descritas no próximo capítulo.

Capítulo 3

Estratégias de Seleção de Caminhos de Teste

A seleção de caminhos para satisfação de critérios estruturais de teste é uma atividade que está incluída em um conjunto maior de procedimentos com o objetivo específico de gerar dados de teste. Abaixo, apresentam-se quatro etapas, extraídas de [3] e [62], que permitem a visualização da inserção da atividade de seleção de caminhos no processo de geração de dados de teste:

1. estabelecimento de elementos requeridos de acordo com algum critério de teste;
2. geração aleatória de caminhos que cubram elementos estabelecidos na etapa anterior;
3. seleção dos caminhos de acordo com alguma estratégia escolhida;
4. geração de dados de teste para cada caminho selecionado.

A atividade de geração de dados de teste é conduzida com o objetivo de encontrar dados de teste que exercitem os elementos requeridos pelos critérios. Para cada programa e elemento requerido por um dado critério de teste estrutural baseado em fluxo de controle e/ou fluxo de dados, existe um conjunto de caminhos de teste, que exercita o elemento requerido.

Isto provoca a necessidade da seleção de um caminho do conjunto de caminhos que cubram um dado elemento requerido para a geração do dado de teste. Várias características podem ser consideradas na seleção de caminhos deste conjunto inicial e que podem permitir aos critérios de teste estrutural maior facilidade de aplicação e maior eficácia e, conseqüentemente, menor

custo. O estudo destas características originou a proposta de estratégias de seleção de caminhos a ser apresentada.

Este capítulo apresenta resultados de um estudo teórico de levantamento de fatores que influenciam a seleção de caminhos, de modo a gerar estratégias de seleção de caminhos a serem aplicadas considerando diferentes características e contextos da atividade de teste de software estrutural.

Na Seção 3.1 é apresentada uma estrutura genérica para representar uma estratégia de seleção de caminhos. Na Seção 3.2 são discutidos diferentes aspectos de software que podem ser considerados para estabelecer estratégias de seleção. Baseadas nestes aspectos, estratégias de seleção de caminhos são propostas e outras estratégias encontradas na literatura são apresentadas. Estas estratégias são descritas utilizando a estrutura proposta na Seção 3.1. Na Seção 3.3 são feitas considerações sobre as estratégias e utilização das mesmas na atividade de teste de software.

3.1 Uma Estrutura de Representação para Estratégias de Seleção de Caminhos

Estratégias de seleção de caminhos de teste têm como objetivo principal selecionar os "melhores" caminhos para a atividade de teste, ou seja, caminhos que tenham maior probabilidade em facilitar essa atividade, seja pela sua complexidade, testabilidade, eficácia ou executabilidade.

De modo a permitir que diferentes características de caminhos possam ser aplicadas a diferentes tipos de problemas, propõe-se uma estrutura geral para estratégias de seleção de caminhos. Utilizando esta estrutura, cada estratégia pode ser representada da mesma forma, permitindo, além da facilidade de apresentação, facilidade de implementação. Esta estrutura é adaptada de [43],[51] e [44]; os dois últimos tratam de medição de complexidade de programas.

A seguir, esta estrutura e um exemplo de utilização são apresentados. Estratégias de seleção de caminhos e o modo de representá-las através desta estrutura são apresentados na próxima seção.

Seja $\Pi_{R,C} = \{\pi_1, \pi_2, ..., \pi_{n-1}, \pi_n\}$ um subconjunto finito de um conjunto de caminhos com-

pletos no grafo G, que cubram o elemento requerido R pelo critério C. Define-se um caminho completo π como uma seqüência finita de nós do grafo G, dada por $(n_1, n_2, ..., n_{j-1}, n_j)$, onde $n_1 = e$, $n_j = s$ e j é o tamanho do caminho.

Outra definição importante é o peso de um caminho. Define-se o peso de um caminho como sendo $P_\pi = \sum_{i=1}^{j}(p(n_i))$, onde n_i é o i-ésimo nó do caminho e $p_s(n_i)$ é o peso do nó n_i, segundo a estratégia de seleção de caminhos S.

Define-se S como o par ordenado $S = (p(n), e(\pi))$, onde $p(n)$ é uma função que atribui pesos aos nós do grafo e $e(\pi)$ uma função de seleção de um caminho π do conjunto $\Pi_{R,C}$. Assim, pode-se definir o conjunto de caminhos selecionados $S\Pi_{R,C}$ pela estratégia S como: $S\Pi_{R,C} = \{\pi \in \Pi_{R,C} \mid e(\pi) = 1\}$, onde e é uma função que possui valor 1 se o caminho deve ser escolhido pela estratégia e 0 caso contrário.

Simplificadamente, baseada na análise do código fonte e do grafo de fluxo de controle e dados do programa, tem-se uma atribuição de pesos a cada nó do grafo. Após esta atribuição, uma somatória dos pesos dos nós de cada caminho é realizada, permitindo-se a escolha do caminho com maior ou menor somatória, ou seja, que possua a característica a ele associada em maior ou menor grau.

O programa a seguir, utilizado em [28], auxilia na exemplificação desta proposta. O programa tem seus blocos de comando divididos, mostrando o número do nó correspondente nas primeiras colunas.

Um exemplo utilizando a Figura 3.1 considera que, por um motivo qualquer, deseja-se selecionar o caminho com menor número de estruturas de controle do tipo "if" e funções "abs()". Define-se que cada estrutura mencionada tem peso 1. A função $e(\pi)$ seleciona o caminho com menor peso total, ou seja, menor P_π. A função $p(n_i)$, que atribui peso ao nó n_i, é a somatória dos pesos das estruturas "if" e "abs()" existentes no nó n_i. Desta forma, os nós (2,3,4,5,7,9,11,12) recebem peso $p = 0$, (1,8,10) recebem peso $p = 1$ e o nó 6, recebe peso $p = 2$, por possuir tanto a estrutura "if" como a função "abs()".

Seja o elemento $R =< 4, (6, 8), \{e\} >$, uma associação requerida pelo critério C = todos-potenciais-usos, o conjunto de caminhos a ser considerado para seleção é: $\Pi_{R,C} = \{(1, 2, 4, 5,$

```
main()
1   {
1     int x, y, pow;
1     float aux, e;
1     float max = 32000.0;
1     if (y>=0)
2       pow = y;
3     else
3       pow = -y;
4     e = 1;
5     while (pow != 0)
6     {
6       aux = e * abs(x);
6       if (aux > max)
7       {
7         printf(" Overflow error!\n");
7         pow = 1;
8       }
8       else
8         e = e *abs(x);
9       pow = pow -1;
9     }
10    if (y<0)
11      e = 1/e;
12    printf("%d\n", e+1);
12  }
```

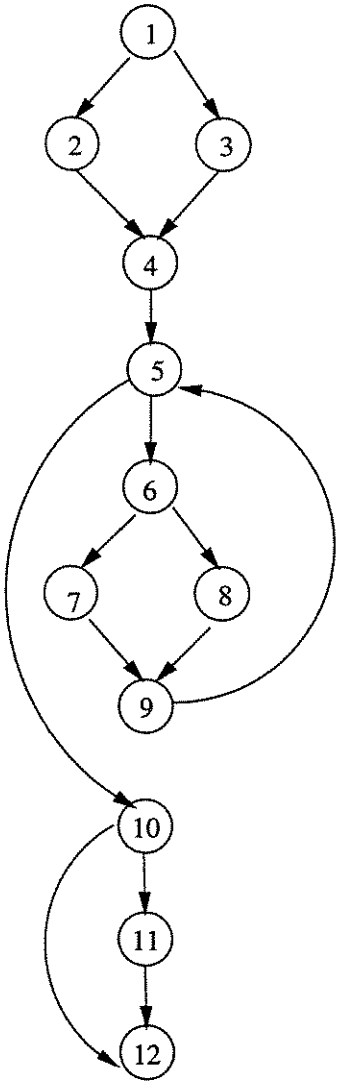

Figura 3.1: Código e grafo de fluxo de controle de um programa exemplo

6, 8, 9, 5, 10, 12),(1, 2, 4, 5, 6, 8, 9, 5, 6, 7, 9, 5, 10, 12),(1, 2, 4, 5, 6, 8, 9, 5, 6, 8, 9, 5, 10, 12)}.

Realizando-se a somatória dos pesos de cada caminho do conjunto $\Pi_{R,C}$, tem-se que o caminho (1, 2, 4, 5, 6, 8, 9, 5, 10, 12) tem peso 5, (1, 2, 4, 5, 6, 8, 9, 5, 6, 7, 9, 5, 10, 12) tem peso 7 e o caminho (1, 2, 4, 5, 6, 8, 9, 5, 6, 8, 9, 5, 10, 12) possui peso 8.

Utilizando a função $e(\pi)$, tem-se que o caminho completo selecionado é (1, 2, 4, 5, 6, 8, 9, 5, 10, 12) com $P_\pi = 5$.

3.2 Estratégias de Seleção de Caminhos

Nesta seção, algumas das principais estratégias de seleção de caminhos encontradas na literatura são apresentadas, e são discutidos diferentes aspectos de software que podem ser considerados para estabelecer e propor novas estratégias de seleção. Cada estratégia é representada utilizando a estrutura proposta na seção anterior, ou seja, como o par ordenado S pode ser instanciado a fim de representar cada estratégia.

Para facilitar o entendimento, exemplos serão apresentados baseados no programa entab.c (Figura 3.2).

```
entab()

1   {
1       int c, col, newcol;
1       int tabstops[MAXLINE];
1       settabs(tabstops);
1       col = 0;
2       do
2       {
2           newcol = col;
3           while((c=getchar()) == BLANK)
4           {
4               newcol++;
4               if(tabpos(newcol,tabstops))
5               {
5                   putchar(TAB);
5                   col = newcol;
5               }
6           }
7           while(col<newcol)
8           {
8               putchar(BLANK);
8               col++;
8           }
9           if(c!=ENDFILE)
9           {
10              putchar(c);
10              if(c==NEWLINE)
11              {
11                  col = 0;
11              }
12              else
12              {
12                  col++;
12              }
13          }
14      }
14      while(c!=ENDFILE);;
15  }
```

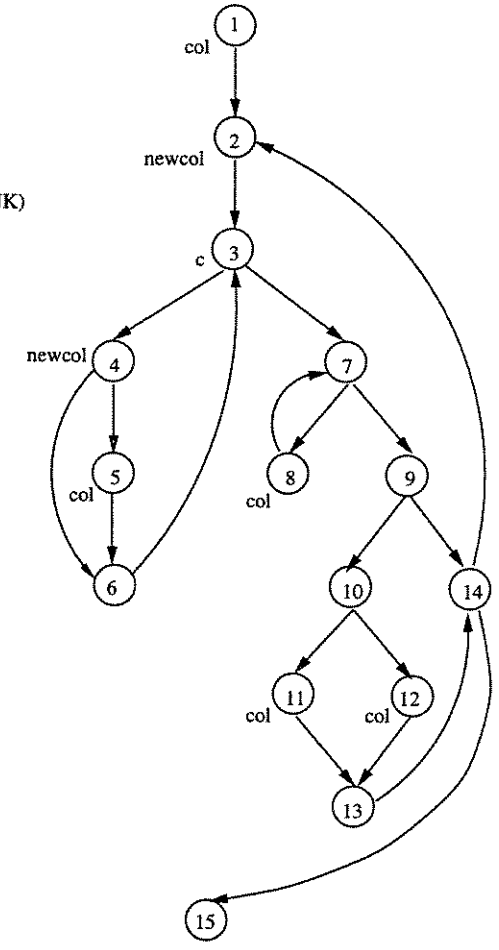

Figura 3.2: Código e grafo de fluxo de controle do programa entab.c

3.2.1 Complexidade de Programas

A complexidade do software, também freqüentemente chamada de *complexidade psicológica*, é referente às características do software que afetam a produtividade do desenvolvedor na composição, compreensão e modificação do software [12], incluindo-se o teste de software. O estudo da complexidade do software recai sobre o estudo das métricas de software já existentes, que têm como objetivo caracterizar quantitativamente certos aspectos do software. Tipos específicos de complexidade são considerados, para a garantia da qualidade de software, incluindo: complexidade do problema, complexidade do projeto (processo) e complexidade do produto. Para cada um destes tipos de complexidade uma métrica apropriada reflete o esforço que um desenvolvedor encontra na execução de tarefas como projeto, codificação, teste ou manutenção de modo a permitir um melhor gerenciamento do desenvolvimento de software, e melhoria na qualidade do produto.

Segundo o CMM (Capability Maturity Model) [10], medições oriundas do desenvolvimento de software são utilizadas para caracterizar o desempenho de novos desenvolvimentos. Para tanto, a utilização de métricas durante o desenvolvimento do software é fundamental para alcançar o 4o. Nível de Maturidade deste modelo, sendo ainda necessária a utilização correta e adaptada de métricas como um meio de estimar custos e auxiliar no planejamento e adaptações tecnológicas, para se alcançar o 5o. Nível de Maturidade.

Métricas de complexidade de software vêm sido desenvolvidas e utilizadas há muito tempo, entre elas Número Ciclomático [35], Extensão do Número Ciclomático [39], *NPath* [40], *Software Science* [19], Taxa de Escopo [20], [25]. Tais métricas continuam sendo desenvolvidas, com características variadas, para serem aplicadas em momentos distintos da atividade de desenvolvimento [37], [42], sobre produtos ou processos [2]. Segundo [12], um subgrupo destas métricas é o de métricas de complexidade de programas, que incluem em seus cálculos o tamanho (como linhas de código), a complexidade da estrutura lógica (fluxo de controle, profundidade do alinhamento, recursão), a complexidade da estrutura de dados (como número de variáveis usadas ou definidas) ou a função (como tipo de software: comercial, científico, distribuído) e ainda combinações destes.

Aplicando-se métricas de complexidade de programas no contexto de seleção de caminhos, tem-se que caminhos mais complexos afetam a produtividade do desenvolvedor, principalmente na compreensão do código para o teste. Pode-se estender este raciocínio para a atividade de geração de dados de teste, uma das mais difíceis do teste de software. Por outro lado, um caminho (ou programa) mais complexo, justamente por representar maior dificuldade de composição, compreensão e manutenção do código, pode representar maior probabilidade de conter um erro ou defeito em seus comandos e estruturas. Isto significa que a seleção de um caminho mais complexo pode ser interessante para o testador, de modo a aumentar a eficácia do teste de software.

Todas as métricas de complexidade de programas utilizadas neste trabalho são largamente conhecidas e utilizadas para a estimação de custos, comparação de produtos e estudos de produtividade de software. É possível aplicá-las logo após a codificação, auxiliando na estimação de custos para a atividade de teste de software.

Propõe-se que elas também sejam utilizadas no contexto de seleção de caminhos. Para isto, foram necessárias adaptações das métricas. Assume-se aqui que o esforço gasto na geração de dados de teste é proporcional ao esforço gasto na codificação do mesmo programa.

Uma observação importante é que deve ser possível ao testador escolher qual tipo de caminho é importante para a atividade de teste a ser realizada: se o caminho com menor ou maior complexidade. Isto porque, como foi apresentado, comandos e estruturas mais complexas contidas em um caminho torna-o mais provável de conter erros, porém, caminhos com menor complexidade facilitam as atividades de geração de teste. Considerando tal facilidade, para o par ordenado $S = (p(n), e(\pi))$, $e(\pi)$ é a função que seleciona o caminho com maior peso, caso se deseje um caminho mais complexo, portanto com maior probabilidade de revelar erros, ou $e(\pi)$ é a função que seleciona o caminho com menor peso, caso se deseje um caminho menos complexo.

LOC

"Uma linha de código é qualquer linha de texto de um programa que não seja um comentário ou uma linha em branco, independentemente do número de comandos ou fragmentos de coman-

dos na linha. Isto inclui especificamente todas as linhas contendo cabeçalhos de programas, declarações e comandos executáveis e não executáveis."[12].

A contagem de linhas de código é uma das métricas de complexidade de programa mais utilizadas e mais fáceis de serem aplicadas [12].

Ainda segundo [12], a quantidade de esforço necessário para construir um programa depende do número total de linhas de código que são escritas. Considera-se o esforço necessário para o teste e suas atividades, como geração de dados de teste, proporcional ao número de linhas de código do programa a ser testado.

A sugestão desta estratégia é selecionar caminhos que contenham os menores números de linhas de código durante a sua execução. A cada bloco de comando, ou seja, nó, tem-se associado o número de linhas de código (LOC) do próprio bloco de comandos. Ao final da somatória dos pesos pode-se selecionar o caminho com menor ou maior peso, de acordo com a observação apresentada anteriormente. Deste modo tem-se, para o par ordenado $S = (p(n), e(\pi))$ para esta estratégia:

$p(n_i)$ =número total de linhas de código do bloco de comando referente ao nó n_i;

De modo a ilustrar esta definição, utiliza-se como exemplo o programa apresentado na Figura 3.2. Considerando $\Pi_{R,C}$ = { (1, 2, 3, 4, 6, 3, 7, 9, 14, 15), (1, 2, 3, 7, 8, 7, 9, 14, 15), (1, 2, 3, 7, 9, 14, 15), (1, 2, 3, 7, 9, 10, 11, 13, 14, 15)}, sendo $R = (14, 15)$, $C = todos - ramos$. Utilizando a estratégia de seleção de caminhos LOC, obtém-se a lista de pesos a seguir, que contém o peso associado a cada nó do grafo de *entab.c*. Após os cálculos de cada P_π tem-se: $P_{\pi_1} = 18$, $P_{\pi_2} = 18$, $P_{\pi_3} = 13$ e $P_{\pi_4} = 20$.

p(1) = 5	p(6) = 1	p(11) = 3
p(2) = 3	p(7) = 1	p(12) = 4
p(3) = 1	p(8) = 4	p(13) = 1
p(4) = 3	p(9) = 1	p(14) = 2
p(5) = 4	p(10) = 3	p(15) = 1

O caminho a ser selecionado neste exemplo é aquele com menor número de linhas de código, ou seja, (1, 2, 3, 7, 9, 14, 15), com $P_{\pi_3} = 13$.

Percebe-se como principal vantagem, a facilidade de implementação desta estratégia, através da automatização do processo de contagem de linhas de código. Esta métrica reflete o esforço de codificação, permitindo a esta estratégia incorporar tal característica também para a fase de teste de software. Algumas desvantagens encontradas nesta estratégia também existem na métrica: sua utilização é dificultada quando a linguagem permite ou exige a utilização de mais de um comando em uma linha; e quando existem linhas com conteúdo mais ou menos complexo, esta complexidade não é refletida por esta métrica e conseqüentemente pela estratégia.

Software Science

Halstead [19] apresentou um modelo do processo de programação, com várias definições associadas, que geraram funções a partir de quatro métricas básicas: $\eta_1, \eta_2, N_1 e N_2$, apresentadas neste texto na subseção Contagem de Tokens. As funções de Halstead mais aceitas [12, 49, 50] são Volume e Esforço e são sugeridas para serem aplicadas na atividade de seleção de caminhos de teste de software.

Contagem de Tokens Uma questão levantada quanto à utilização de LOC é que umas linhas são mais difíceis de codificar que outras. Uma solução para este problema é considerar mais items em uma linha, que resulta na contagem de tokens, que é uma unidade sintática básica distinguível por um compilador. Halstead [19] propõe a contagem de tokens como um dos prodecimentos para a aplicação de sua métrica Software Science e Conte *et al.* [12] propõem a utilização direta dos tokens como uma métrica de medição de complexidade.

Para a realização desta contagem, que é dependente de linguagem, um programa é uma coleção de tokens que podem ser classificados em operadores e operandos. Geralmente, qualquer símbolo ou palavra-chave em um programa que especifica uma ação é considerado um operador, enquanto um símbolo usado para representar dado é considerado operando. Muitas marcas de pontuação são categorizadas como operadores. Variáveis, constantes, e mesmo rótulos (*labels*) são operandos. Operadores consistem de símbolos aritméticos (tais como +, -, e /), nomes de comandos (tais como WHILE, FOR e READ), símbolos especiais (tais como {}, [], $->$) e mesmo nomes de funções (tais como EOF, EXIT).

As métricas básicas utilizadas são:

η_1 = número de operadores únicos;

η_2 = número de operandos únicos;

N_1 = número de ocorrências de operadores;

N_2 = número de ocorrências de operandos;

O tamanho de um programa em termos do número total de tokens usado é:

$N = N_1 + N_2$

A utilização desta métrica como estratégia de seleção de caminhos se dá com a contagem de tokens sendo realizada para cada bloco de comando e os valores das métricas básicas sendo associadas como pesos aos nós correspondentes.

Deste modo, para o par ordenado $S = (p(n), e(\pi))$ para esta estratégia:

$p(n_i) = N_i$, onde N_i é o número total de tokens executados no bloco de comando referente ao nó n_i.

Abaixo, são apresentados os pesos associados a cada nó, a partir da função $p(n_i)$ para esta estratégia e para o programa-exemplo da Figura 3.2.

p(1) = 23	p(6) = 1	p(11) = 6
p(2) = 6	p(7) = 6	p(12) = 9
p(3) = 12	p(8) = 13	p(13) = 1
p(4) = 16	p(9) = 6	p(14) = 9
p(5) = 11	p(10) = 12	p(15) = 1

Considerando: $\Pi_{R,C}$ = {(1, 2, 3, 7, 9, 14, 15),(1, 2, 3, 7, 8, 7, 9, 14, 15),(1, 2, 3, 4, 5, 6, 3, 7, 9, 14, 15),(1, 2, 3, 7, 8, 7, 8, 7, 9, 14, 15) }, ou seja, um subconjunto finito de caminhos completos que satisfazem o elemento $R = (9, 14)$ requerido pelo critério $C = todos - ramos$.

Obtém-se para cada caminho π, a somatória total dos pesos P_π: $P_{\pi_1} = 63$, $P_{\pi_2} = 82$, $P_{\pi_3} = 103$ e $P_{\pi_4} = 101$.

Como se deseja neste exemplo o caminho com maior número de tokens, para possibilitar que caminhos mais complexos sejam testados, tem-se que o caminho π_3 é o caminho selecionado.

A facilidade de automatização é novamente uma vantagem. Esta métrica, segundo [12], é mais completa que LOC, ou seja, tem maior probabilidade de se aproximar mais do esforço de codificação, inclusive do teste.

Volume do Programa Halstead sugere que o volume de um programa, alcançado a partir de:

$V = N \times log_2 \eta$

pode ser interpretado como o número de comparações mentais necessárias para escrever um programa de tamanho N. Esta interpretação assume que, durante a programação, a mente humana assume um processo de "busca binária" para selecionar o próximo token a partir do vocabulário de um programa, de tamanho η, onde vocabulário é $\eta = \eta_1 + \eta_2$.

Esta métrica pode ser adaptada à estrutura proposta para estratégia de seleção de caminhos, através do par ordenado $S = (p(n), e(\pi))$, onde :

$p(n_i) = V_i$, onde $V_i = V$ limitado ao escopo de contagem de tokens para o bloco de comando n_i, ou seja: N e η são contagens de tokens contidos apenas no bloco de comando n_i;

Um exemplo, utilizando o programa apresentado na Figura 3.2, ilustra a aplicação desta estratégia. A seguir são apresentados os pesos associados a cada nó, a partir da função $p(n_i)$ para esta estratégia.

p(1) = 92	p(6) = 0	p(11) = 15,50
p(2) = 15,50	p(7) = 15,50	p(12) = 27
p(3) = 36	p(8) = 44,97	p(13) = 0
p(4) = 57,36	p(9) = 15,50	p(14) = 27
p(5) = 36,54	p(10) = 38,04	p(15) = 0

Considerando: $\Pi_{R,C}$ = {(1, 2, 3, 4, 5, 6, 3, 7, 8, 7, 8, 7, 8, 7, 9, 14, 15), (1, 2, 3, 7, 8, 7, 9, 10, 12, 13, 14, 15), (1, 2, 3, 7, 8, 7, 9, 10, 11, 13, 14, 15), (1, 2, 3, 4, 6, 3, 7, 8, 7, 9, 14,

15) }, ou seja, um subconjunto finito de caminhos completos que satisfazem o elemento R = $< 3, (7, 8), \{c\} >$ requerido pelo critério C = todos-potenciais-usos.

Obtém-se para cada caminho π, a somatória total dos pesos P_π: $P_{\pi_1} = 512,81$, $P_{\pi_2} = 327,01$, $P_{\pi_3} = 315,51$ e $P_{\pi_4} = 355,33$.

Como se deseja neste exemplo o caminho com menor volume, para reduzir a complexidade de geração de dados de teste, tem-se que o caminho π_3 é o menos complexo.

Esforço Halstead [19] propôs que o esforço requerido para implementar um programa de computador aumenta à medida que o tamanho do programa aumenta. Também é gasto mais esforço para implementar um programa em mais baixo nível do que outro programa equivalente em mais alto nível. Ainda, o esforço E em *Software Science* é definido como:

$E = (\eta_1 N_2 N log_2 \eta)/2\eta_2$

A unidade de medida E representa as discriminações mentais elementares.

Esta métrica pode ser adaptada ao modelo de estratégia de seleção de caminhos proposto, onde $p(n)$, do par ordenado $S = (p(n), e(\pi))$, assume o seguinte valor:

$p(n_i) = E_i$, onde $E_i = E$ que possui os argumentos que contribuem para este cálculo limitados ao escopo de contagem de tokens para o bloco de comando n_i, ou seja: $\eta_1, \eta_2, N_1 e N_2$ são contagens de tokens contidos apenas no bloco de comando n_i;

A seguir são apresentados os pesos associados a cada nó, a partir da função $p(n_i)$ para esta estratégia e para o programa-exemplo da Figura 3.2.

p(1) = 532,28	p(6) = 0	p(11) = 31,02
p(2) = 31,02	p(7) = 31,02	p(12) = 121,5
p(3) = 108	p(8) = 239,85	p(13) = 0
p(4) = 344,16	p(9) = 31,02	p(14) = 121,5
p(5) = 127,89	p(10) = 199,70	p(15) = 0

O subconjunto finito de caminhos completos a sofrer a seleção é: $\Pi_{R,C}$ = { (1, 2, 3, 7, 8, 7, 9, 10, 11, 13, 14, 2, 3, 4, 6, 3, 7, 9, 14, 15), (1, 2, 3, 7, 9, 10, 11, 13, 14, 2, 3, 4, 6, 3, 7, 8, 7, 9, 14, 15), (1, 2, 3, 7, 9, 10, 11, 13, 14, 2, 3, 4, 6, 3, 7, 9, 10, 12, 13, 14, 15)}.

Estes caminhos cobrem o elemento R = $< 11, (6, 3), \{col\} >$ requerido pelo critério C = todos-potenciais-usos.

Obtém-se para cada caminho π, a somatória total dos pesos P_π: $P_{\pi_1} = 2130,87$, $P_{\pi_2} = 2130,87$ e $P_{\pi_3} = 2181,48$.

Como se deseja, neste exemplo, o caminho com menor esforço, para reduzir a complexidade de geração de dados de teste, tem-se os caminhos π_1 e π_2.

No contexto do teste de software, uma atividade realizada após a codificação, de modo que o programa a ser contado está pronto, estas métricas podem auxiliar na estimação de custo e volume para a sub-atividade de geração de dados de teste. A geração de dados de teste se utiliza de comparações mentais (e comparações, na execução simbólica) para encontrar os dados de entrada que executem os caminhos de teste.

Outras métricas de *Software Science* [19] podem ser melhor estudadas e também utilizadas como estratégias de seleção de caminhos, como Vocabulário, Tamanho Estimado, Volume Potencial, Dificuldade, Tempo e Nível de Linguagem. Um trabalho empírico pode ser conduzido para a verificação de qual das métricas de Software Science podem ser mais úteis quanto à facilidade de geração de dados de teste e quanto à eficácia do teste e outra para a previsão de esforço na geração de dados de teste.

Quantidade de Definições e Usos de Variáveis

Segundo Conte *et al.* [12], um programador deve estar constantemente consciente da situação de um número de itens de dados durante o processo de programação. Uma hipótese razoável é a de que quanto mais itens de dados um programador deva manter sob controle, quando da construção de um comando, mais difícil será construí-lo.

Portanto o interesse desta estratégia de seleção de caminhos é permitir a escolha de um caminho mais simples quanto ao número de definições e usos de variáveis ou um mais complexo, aumentando a probabilidade de se ocorrer uma falha.

Aqui, sugerem-se três tipos de estratégias, baseadas nas definições e usos de variáveis, onde $p(n)$ do par ordenado $S = (p(n), e(\pi))$ assume os valores, respectivamente:

- $p(n_i)$ = número total de ocorrências de **definições** de variáveis no nó n_i, de tal modo que se selecione o caminho com menor ou maior número de definições de variáveis;
- $p(n_i)$ = número total de ocorrências de **usos** de variáveis no nó n_i , de modo a selecionar o caminho com menor ou maior número de usos; e
- $p(n_i)$ = número total de ocorrências de **definições e usos** de variáveis no nó n_i, selecionando o caminho com menor ou maior número de definições e usos.

Abaixo, são apresentados os pesos associados a cada nó, a partir da função $p(n_i)$, referente à primeira estratégia apresentada, ou seja, para a estratégia definições de variáveis, para o programa-exemplo da Figura 3.2.

p(1) = 1	p(6) = 0	p(11) = 1
p(2) = 1	p(7) = 0	p(12) = 1
p(3) = 1	p(8) = 1	p(13) = 0
p(4) = 1	p(9) = 0	p(14) = 0
p(5) = 1	p(10) = 1	p(15) = 0

O conjunto de caminhos a considerar para a seleção é: $\Pi_{R,C}$ = { (1, 2, 3, 4, 6, 3, 7, 9, 14, 15), (1, 2, 3, 4, 5, 6, 3, 7, 9, 14, 15), (1, 2, 3, 4, 6, 3, 7, 8, 7, 9, 14, 15), (1, 2, 3, 4, 6, 3, 7, 9, 10, 11, 13, 14, 15)}, ou seja, um subconjunto finito de caminhos completos que satisfazem o elemento R = $< 2, (3, 4), \{newcol\} >$ requerido pelo critério C = todos-potenciais-usos.

Obtém-se para cada caminho seu peso total: $P_{\pi_1} = 5$, $P_{\pi_2} = 6$, $P_{\pi_3} = 6$ e $P_{\pi_4} = 7$.

Como se deseja neste exemplo o caminho com maior número de variáveis definidas, para possibilitar que caminhos mais complexos sejam testados, tem-se que o caminho π_4 é o mais complexo.

A seleção de caminhos que possuam menor número de definições e/ou usos é sentida intuitivamente na geração de dados de teste, permitindo a diminuição do esforço de geração de dados de teste. Mas é interessante notar que um caminho que possua maior número de definições e usos de variáveis pode ter maior probabilidade de conter erros, sendo viável a seleção de caminhos que possuam o maior número de definições e/ou usos de variáveis, de modo que o teste possa tornar-se mais eficaz.

Nível de Aninhamento

Dunsmore e Gannon [13] apresentam o nível de aninhamento de programas como algo importante a ser considerado na medição de complexidade de programas. Quanto maior a profundidade ou nível de aninhamento, mais difícil é avaliar as condições de entrada para certos comandos. De modo a computar esta métrica, Conte *et al.* [12] sugerem que cada comando executável em um programa deve ter um nível de aninhamento a ele atribuído. Um procedimento recursivo para realizar tal atribuição é descrito a seguir:

1. Ao primeiro comando executável é atribuído o nível de aninhamento 1;
2. Se o comando a está no nível l e o comando b simplesmente segue seqüencialmente a execução de um comando a, então o nível de aninhamento do comando b é l também;
3. Se o comando a está no nível l e o comando b está dentro de um laço ou uma transferência condicional governada pelo comando a, então o nível de aninhamento de b é $l + 1$.

Esta métrica pode ser adaptada de modo que cada bloco de comando, que nem sempre possui todos os seus comandos no mesmo nível de aninhamento, assuma o nível de aninhamento do comando com maior l. O nível de aninhamento de cada bloco é associado ao peso de cada nó correspondente. Desta maneira, tem-se para o par ordenado $S = (p(n), e(\pi))$:

$p(n_i)$ = nível de aninhamento l do comando com maior l contido no bloco de comando referente ao nó n_i, sendo l determinado segundo o procedimento apresentado;

Desta maneira, pretende-se gerar uma estratégia que selecione o caminho que possua comandos num maior ou menor nível de aninhamento.

Abaixo são apresentados os pesos associados a cada nó, a partir da função $p(n_i)$ para esta estratégia e para o programa-exemplo da Figura 3.2.

Considerando: $\Pi_{R,C}$ = { (1, 2, 3, 7, 9, 14, 15), (1, 2, 3, 7, 8, 7, 9, 14, 15), (1, 2, 3, 4, 5, 6, 3, 7, 9, 14, 15), (1, 2, 3, 7, 8, 7, 8, 7, 9, 14, 15)}, caminhos completos que cobrem o elemento R = $< 3, (9, 14), \{c\} >$ requerido pelo critério C = todos-potenciais-usos. Obtém-se para cada caminho π, a somatória total dos pesos P_π: $P_{\pi_1} = 12$, $P_{\pi_2} = 17$, $P_{\pi_3} = 24$ e $P_{\pi_4} = 22$.

p(1) = 1	p(6) = 3	p(11) = 4
p(2) = 2	p(7) = 2	p(12) = 4
p(3) = 2	p(8) = 3	p(13) = 3
p(4) = 3	p(9) = 2	p(14) = 2
p(5) = 4	p(10) = 3	p(15) = 1

Como se deseja, neste exemplo, o caminho com maior número de nós com nível de aninhamento maior, para possibilitar que caminhos mais complexos sejam testados, seleciona-se o caminho π_3.

Métrica de Silva e Price[29]

Em [51], foi proposta uma métrica considerando que as métricas até então não conseguiam capturar a influência de estruturas de controle diferentes na complexidade de entendimento do programa. Esta métrica possui como base para contagem o programa fonte e caminhos requeridos pelo critério de teste todos-du-caminhos [48].

Em [44], a métrica proposta por Silva e Price é instanciada com o critério de seleção de caminhos todos-potenciais-du-caminhos [28]: " Seja $(s_1, ..., s_k)$ o conjunto de subcaminhos $(n_i, ..., n_j)$ selecionados de acordo com o critério todos-potenciais-du-caminhos, proposto em [28]".

A complexidade de um programa é calculada utilizando-se a seguinte fórmula:

$$\sum_{y=1}^{k} \left(\left(\frac{\sum_{x=1}^{j} (b(n_{zy}))}{j - i + 1} \right) \times (\#e + 1) \right)$$

onde $b(n_{zy})$ indica a complexidade bruta no nó n_z do subcaminho y e $\#e$ indica o número de estruturas diferentes representadas pelos nós do subcaminho.

Chega-se à complexidade bruta de um nó atribuindo um peso a cada nó, de acordo com a estrutura de controle que ele representa. A atribuição dá-se da seguinte maneira:

- nó representando o teste de uma estrutura de controle de seleção (IF): 2;
- nó representando a condição de uma estrutura de controle de repetição do tipo WHILE ou FOR: 3;

- nó representando o teste de uma estrutura de controle de repetição do tipo REPEAT-UNTIL:4;
- nó representando o teste de uma variável em uma estrutura de controle do tipo CASE: número de alternativas do CASE. O peso bruto de cada alternativa desta estrutura é a soma dos pesos brutos das estruturas contidas na alternativa em questão;
- nó representando um desvio incondicional de controle (GOTO):8.
- nó contendo um bloco básico (comandos seqüenciais):1;

Ainda segundo [51], no caso de haver subcaminhos contidos em outros, são considerados, para este cálculo, somente os subcaminhos que os contém, a fim de não considerar mais de uma vez determinadas relações entre a definição e o uso de determinada variável. Um exemplo de como a métrica é aplicada encontra-se em [51].

A estratégia de seleção de caminhos adaptada a partir desta métrica é simplificada, possuindo, a mesma atribuição de pesos sugerida por Silva e Price. Ou seja, para o par ordenado $S = (p(n_i), e(\pi))$, onde $p(n_i)$ é a *complexidade bruta* no nó n_i.

A seguir, são apresentados os pesos associados a cada nó, a partir da função $p(n_i)$ para esta estratégia e para o programa-exemplo da Figura 3.2.

p(1) = 1	p(6) = 0	p(11) = 1
p(2) = 1	p(7) = 3	p(12) = 1
p(3) = 3	p(8) = 1	p(13) = 0
p(4) = 2	p(9) = 2	p(14) = 4
p(5) = 1	p(10) = 2	p(15) = 0

Considerando: $\Pi_{R,C}$ = { (1, 2, 3, 7, 9, 14, 2, 3, 7, 9, 14, 15), (1, 2, 3, 7, 9, 14, 2, 3, 7, 8, 7, 9, 14, 15), (1, 2, 3, 7, 8, 7, 9, 14, 2, 3, 7, 9, 14, 15), (1, 2, 3, 7, 9, 14, 2, 3, 7, 9, 10, 12, 13, 14, 15), (1, 2, 3, 7, 9, 14, 2, 3, 7, 9, 10, 11, 13, 14, 15)}, que cobre o elemento R = (14, 2) requerido pelo critério C = todos-ramos.

Obtém-se para cada caminho π, a somatória total dos pesos P_π: $P_{\pi_1} = 27$, $P_{\pi_2} = 31$, $P_{\pi_3} = 31$, $P_{\pi_4} = 30$ e $P_{\pi_5} = 30$.

Como se deseja, neste exemplo, o caminho com menor complexidade, de modo a reduzir a complexidade da geração de dados de teste, tem-se que o caminho π_1 é o selecionado.

3.2.2 Menor Número de Predicados

Considerando a questão da não executabilidade, em [62] e [32], Malevris *et al.* apresentam um estudo para minimizar os efeitos causados por caminhos não executáveis no teste de ramos e conseqüentemente no tempo e no custo desta atividade. Como apresentado no Capítulo 2, eles validam a seguinte hipótese: quanto maior o número de predicados em um caminho, menor a probabilidade de ele ser executável. Vergilio *et al.* [53] validaram os resultados de Malevris *et al.* no contexto de critérios baseados em fluxo de dados. Em seus trabalhos, Bertolino e Marré [3], utilizam a hipótese de Malevris como uma estratégia de seleção de caminhos.

Neste texto, apresenta-se a representação dessa estratégia segundo a estrutura da Seção 3.1. Para o par ordenado $S = (p(n), e(\pi))$:

$p(n_i)$ = número de predicados do bloco de comando correspondente ao nó n_i;

$e(\pi)$ = é a função que escolhe o caminho com menor peso total P_π.

Abaixo são apresentados os pesos associados a cada nó, a partir da função $p(n_i)$ para esta estratégia e para o programa-exemplo da Figura 3.2.

p(1) = 0	p(6) = 0	p(11) = 0
p(2) = 0	p(7) = 1	p(12) = 0
p(3) = 1	p(8) = 0	p(13) = 0
p(4) = 1	p(9) = 1	p(14) = 1
p(5) = 0	p(10) = 1	p(15) = 0

Considerando o elemento R = $< 1, (14, 15), \{col\} >$ requerido pelo critério C = todos-potenciais-usos e o conjunto $\Pi_{R,C}$ = {(1, 2, 3, 4, 6, 3, 4, 6, 3, 7, 9, 14, 15), (1, 2, 3, 7, 9, 14, 15), (1, 2, 3, 4, 6, 3, 7, 9, 14, 15)} um conjunto finito de caminhos que cobrem o elemento R.

O caminho (1, 2, 3, 7, 9, 14, 15) é o caminho selecionado pela estratégia "Menor Número de Predicados", pois apresenta o menor número de predicados: $P_{\pi_2} = 4$; enquanto o caminho (1, 2, 3, 4, 6, 3, 4, 6, 3, 7, 9, 14, 15) possui $P_{\pi_1} = 8$ e (1, 2, 3, 4, 6, 3, 7, 9, 14, 15), $P_{\pi_3} = 6$.

Esta teoria, transformada em estratégia de seleção de caminhos propicia a seleção de caminhos que possuam maior probabilidade de serem executáveis. Isto é fundamental na atividade de geração de dados de teste, que sofre grande impacto nos custos devido à indecidibilidade sobre a executabilidade de um caminho [47].

3.2.3 Testabilidade

A testabilidade, segundo [57], é determinada pela estrutura do código, semântica e distribuição de entrada do programa. Um programa possui alta testabilidade quando prontamente são detectados defeitos através de testes funcionais; um programa com baixa testabilidade é aquele que possui baixa probabilidade de revelar defeitos no teste funcional de entrada aleatória.

Voas *et al.* [57] introduziram a Análise de Sensibilidade (Sensitivity Analysis), onde "sensibilidade" significa uma predição da probabilidade de que um defeito irá causar uma falha no software em uma localização particular sob uma distribuição específica de entrada.

Um procedimento da análise de sensibilidade é a análise sucessiva da execução do programa, infecção e propagação. Na análise de execução, são requeridos uma distribuição de entrada específica, a execução do código com entradas aleatórias a partir daquela distribuição e o registro de localizações do código executado para cada entrada. Ainda, a análise de execução é referente à probabilidade de uma localização em particular afetar a saída. A análise de infecção requer a criação de código mutante, ou seja, cópias do código original que tenham sido sintaticamente alteradas. Depois de criar um conjunto de mutantes, é obtida uma estimativa de probabilidade de que o estado do dado é afetado pela presença de cada mutante do conjunto. Uma estimativa de infecção é uma função de uma localização, o mutante criado e o conjunto de estados de dados que ocorrem antes da localização. A análise de propagação estima a probabilidade de que um estado de dados infectado em uma localização irá se propagar para a saída. Esta análise é baseada nas alterações causadas pelos estados dos dados, utilizando uma função perturbação.

Propõe-se como estratégia de seleção de caminhos uma parte da proposição de Voas *et al.* [57], a análise de execução, considerada a parte mais robusta da Análise de Sensibilidade.

Para permitir a aplicação desta estratégia, dados de teste devem ser gerados aleatoriamente

e aplicados ao programa. Os nós alcançados pelos dados de teste devem ser registrados em conjunto com o número de vezes que foram alcançados. Na seleção de caminhos, são escolhidos os que possuem nós que tiveram menor média de alcance registrada na etapa anterior.

Desta forma, estabelece-se um conjunto Δ de nós com a menor média de alcance. Define-se para o par ordenado $S = (p(n), e(\pi))$:

$p(n_i) = 0$, se $i \notin \Delta$ e

$p(n_i) = 1$, se $i \in \Delta$.

$e(\pi)$ é a função que escolhe o caminho com maior peso total P_π.

A lista abaixo apresenta os pesos associados a cada nó, a partir da função $p(n_i)$ para esta estratégia e para o programa-exemplo da Figura 3.2, considerando $\Delta = \{5, 11\}$.

p(1) = 0	p(6) = 0	p(11) = 1
p(2) = 0	p(7) = 0	p(12) = 0
p(3) = 0	p(8) = 0	p(13) = 0
p(4) = 0	p(9) = 0	p(14) = 0
p(5) = 1	p(10) = 0	p(15) = 0

Considerando: $\Pi_{R,C} = \{$(1, 2, 3, 4, 6, 3, 7, 9, 14, 2, 3, 4, 6, 3, 7, 9, 14, 2, 3, 4, 6, 3, 7, 8, 7, 9, 10, 12, 13, 14, 15), (1, 2, 3, 7, 9, 14, 2, 3, 7, 9, 14, 2, 3, 4, 6, 3, 4, 5, 6, 3, 4, 5, 6, 3, 7, 8, 7, 9, 10, 12, 13, 14, 15), (1, 2, 3, 7, 9, 14, 2, 3, 7, 9, 14, 2, 3, 4, 6, 3, 4, 5, 6, 3, 4, 6, 3, 7, 8, 7, 9, 10, 11, 13, 14, 15), (1, 2, 3, 4, 6, 3, 4, 5, 6, 3, 7, 9, 14, 2, 3, 7, 9, 10, 12, 13, 14, 15), (1, 2, 3, 7, 9, 14, 2, 3, 7, 9, 14, 2, 3, 4, 6, 3, 4, 5, 6, 3, 7, 9, 14, 15)$\}$, caminhos que cobrem o elemento $< 1, (4, 6), \{col\} >$ requerido pelo critério C = todos-potenciais-usos.

Obtém-se a somatória total dos pesos dos caminhos: $P_{\pi_1} = 0$, $P_{\pi_2} = 2$, $P_{\pi_3} = 2$, $P_{\pi_4} = 1$ e $P_{\pi_5} = 1$.

De acordo com $e(\pi)$ os caminhos selecionados são π_2 e π_3.

A análise de execução avalia o grau em que comandos foram executados a partir da entrada de um conjunto aleatório de dados de teste. Desta maneira, procura-se selecionar caminhos que possuam blocos de comandos que tenham sido *pouco* executados. O que se espera com esta característica é propiciar a geração de dados de teste para a execução destes caminhos

selecionados, de modo a aumentar a probabilidade de se passar por um defeito, que pode estar nestes blocos de comandos menos executados, e causar uma falha.

Algumas desvantagens aparecem, quando da análise desta estratégia de seleção de caminhos. A primeira é que não existe trabalho aplicando a proposta inteiramente. A segunda é que Bertolino e Strigini [5] questionam algumas definições dadas por Voas *et al.*. No entanto, percebe-se que este tipo de estratégia de seleção de caminhos tem sua utilidade para a atividade de teste quando são exigidos alcançar nós que tenham pouca probabilidade de serem alcançados.

3.2.4 Outras Estratégias

Uma estratégia que pode ser utilizada com a estrutura geral proposta na Seção 3.1 e que se preocupa com a eficácia do teste de software, é uma que considera as classes de erros. Para esta estratégia sugere-se que, de posse de classes de erros que possam ser mapeadas para comandos, ou seja, para blocos de comandos, é possível selecionar caminhos que tenham maior probabilidade de revelar defeitos de determinada classes de erros.

Vários autores apresentaram abordagens que podem ser enquadradas como estratégias de seleção de caminhos, mas que, no contexto deste trabalho, não podem ser representadas utilizando a estrutura geral proposta na Seção 3.1. Vergilio [55] introduziu critérios de teste baseados em restrições, ou seja, dado um conjunto de caminhos que satisfazem um determinado critério estrutural, os dados de teste selecionados para executar estes caminhos são tais que, durante a execução do caminho, uma restrição é satisfeita. Essa restrição descreve erros específicos e aumenta a probabilidade de revelar defeitos.

Bertolino e Marré, em [3] e [4], introduzem a estratégia de selecionar o menor conjunto de caminhos que satisfazem os critérios todos-ramos e todos-usos, através do uso de arcos e *duas* irrestritos, minimizando o número de casos de teste. As definições de arcos e *duas* irrestritos foram apresentadas na Seção 2.4.

3.3 Considerações Finais

As estratégias de seleção de caminhos são utilizadas para selecionar os "melhores" caminhos, a partir de um conjunto inicial de caminhos que cobrem um dado elemento requerido por algum critério de teste (adequação e/ou seleção). Nenhuma análise específica de cobertura de teste do programa é realizada para as estratégias de seleção. Tais estratégias são criadas para que os critérios de teste estrutural tenham maior facilidade de aplicação, sem que a qualidade do teste de software seja comprometida pela ausência dos critérios.

Análoga à teoria dos grafos, onde os custos associados a arestas podem representar distância, valores monetários ou pesos, a estrutura de representação de estratégias de seleção de caminhos definida neste capítulo propõe que os custos sejam associados a complexidade, testabilidade e não executabilidade, ou ainda, a tempo, esforço ou custo do teste de software. Desta forma, sugere-se que o problema fique reduzido a selecionar o maior ou menor caminho que traduza a necessidade do testador. Por exemplo, considerando a complexidade do software, podem ser escolhidos caminhos mais simples ou mais complexos. Intuitivamente, caminhos mais simples podem ser utilizados para facilitar a atividade de geração de dados de teste e caminhos mais complexos possuem maior probabilidade de conter defeitos, aumentando a eficácia do teste de software.

A Tabela 3.1 permite visualizar o conjunto de estratégias proposta neste texto, que instanciam a estrutura geral definida pelo par ordenado $S = (p(n), e(\pi))$.

Ainda, as estratégias propostas não são exclusivas. Estas podem ser combinadas, podendo reduzir o número de caminhos não executáveis através da associação das diversas estratégias com a estratégia menor número de predicados. Isto é possível através da soma proporcional de pesos, permitida pela idéia da estrutura geral proposta. Para isto, é necessária a criação de outra função $p(n)$ do par ordenado S ou de uma tupla $T = (p(n), q(n), e_1(\pi), e_2(\pi))$, o que proporcionaria um critério de desempate na seleção de caminhos como um meio de aumentar a probabilidade de selecionar um caminho executável. Apenas um trabalho prático mostraria a validade desta proposta.

Um observação importante é que apesar das estratégias terem sido ilustradas utilizando os

Tabela 3.1: Estratégias de Seleção de Caminhos

Estratégias	$p(n_i)$	$e(\pi)$
LOC	número total de linhas de código do nó n_i	maior ou menor P_π
Contagem de Tokens	N_i, onde N_i é o número total de tokens do nó n_i	maior ou menor P_π
Volume	$V_i = N_i \times log_2\eta_i$	maior ou menor P_π
Esforço	$E_i = (\eta_{1_1} N_{2_1} N_1 log_2\eta_1)/2\eta_{2_1}$	maior ou menor P_π
Quant.Def. e Usos de Var.	número total de ocorrências de **definições**, **usos** ou **definições e usos** de variáveis no nó n_i	maior ou menor P_π
Nível de Aninhamento	nível de aninhamento l do comando com maior l contido no nó n_i	maior ou menor P_π
Métrica de Silva e Price	*complexidade bruta* no nó n_i	maior ou menor P_π
Menor No. de Predicados	número de predicados do nó n_i	menor P_π
Testabilidade	$p(n_i) = 0$, se $i \notin \Delta$ e $p(n_i) = 1$, se $i \in \Delta$	maior P_π

critérios todos-potenciais-usos e todos-ramos, estas podem ser utilizadas para satisfazer qualquer critério baseado em caminhos, como por exemplo todos-nós, todas-condições [45] ou todos-usos [47, 59, 48].

Segundo Bertolino e Marré [3], e com objetivos de redução de custos, estratégias de seleção de caminhos devem incorporar meios de minimizar a seleção de caminhos não executáveis de modo a facilitar a geração de dados de teste. Considerando estes objetivos, e de modo a avaliar a estratégia menor número de predicados, foi implementado um módulo de apoio à ferramenta de teste Poke-Tool, discutido no Capítulo a seguir. A implementação do módulo veio validar a estrutura de representação para estratégias de seleção de caminhos como um mecanismo prático e poderoso de automatização de estratégias.

Capítulo 4

O Módulo Poke-Paths

Um módulo de geração de caminhos, o Poke-Paths, foi desenvolvido implementando a estrutura de representação de estratégias de seleção de caminhos proposta no Capítulo 3. Este módulo é integrado à ferramenta de teste Poke-Tool e gera caminhos completos tais que exercitem um dado elemento requerido pelos critérios todos-nós, todos-ramos ou por critérios da família Potenciais Usos.

Este capítulo detalha a implementação do módulo Poke-Paths, através da definição do problema a ser resolvido, a contextualização do módulo no teste de software, explicando a solução adotada e o funcionamento geral do mesmo.

4.1 Descrição do Problema

No teste estrutural, satisfazer um critério de teste significa exercitar todos os elementos estruturais requeridos por ele. Para que isto ocorra, é necessário selecionar dados de forma que cada elemento requerido seja exercitado pela execução de um caminho completo do programa. A escolha deste caminho completo e a seleção do conjunto de dados para executá-lo são tarefas da geração de dados de teste. A geração automática de caminhos completos é a primeira etapa a ser cumprida para automatizar tal atividade.

Considerando o conjunto de caminhos que cubram um elemento requerido, deve-se gerar caminhos completos automaticamente, selecionando alguns caminhos deste conjunto, para que a partir deles haja a geração de dados de teste. Estratégias de seleção de caminhos como as

apresentadas no Capítulo 3, constituem a base para a geração automática de caminhos. A seguir são apresentados alguns requisitos do módulo desenvolvido:

1. A geração automática de caminhos completos deve ser baseada nas estratégias de seleção de caminhos, definidas no Capítulo 3. Ou seja, deve ser implementada a estrutura de representação de estratégias de seleção de caminhos, o par ordenado $S = (p(n), e(\pi))$;

2. Implementar mecanismos para reduzir o número de caminhos completos não executáveis passados à atividade de geração de dados de teste. Um dos meios é implementar a estratégia menor número de predicados, permitindo a geração de caminhos com maior probabilidade de serem executáveis. Segundo Bertolino e Marré [3] esta é uma necessidade de ferramentas de geração automática de caminhos;

3. O módulo Poke-Paths deve propiciar a geração automática de caminhos completos que cubram elementos requeridos pelos critérios todos-nós, todos-arcos e pela Família de Critérios Potenciais Usos [28]. Entende-se por elementos um conjunto não vazio de elementos associados a um grafo de fluxo de controle e fluxo de dados, como nó, arco, potencial-associação e potencial-du-caminho, descritos no Capítulo 2;

4. Deve ser implementada a geração automática de caminhos aleatórios. Este tipo de geração de caminhos não é baseada em estratégias de seleção de caminhos e não é possível representá-la pelo par ordenado S. No entanto a geração aleatória de caminhos é implementada de modo a facilitar a condução de futuros experimentos a respeito do teste de software estrutural.

4.2 Contexto do Módulo

Como apresentado no Capítulo 2, a ferramenta de teste Poke-Tool implementa a família de critérios Potenciais Usos e, entre outras funcionalidades, determina os elementos requeridos por estes critérios. O módulo de apoio Poke-Paths foi desenvolvido para apoiar a atividade de teste gerando caminhos a partir desses elementos requeridos, dando suporte às estratégias de seleção de caminhos apresentadas no Capítulo 3.

A Figura 4.1 ilustra o contexto no qual está inserida a geração de caminhos completos em uma ferramenta de teste estrutural, a partir de um procedimento qualquer em um programa. A partir dos caminhos completos gerados na aplicação do módulo Poke-Paths, o testador gera, manualmente ou através de uma ferramenta automática, um conjunto de dados de teste que executem o conjunto de caminhos completos gerados. O resultado da execução deste programa com os dados encontrados são comparados com os resultados esperados.

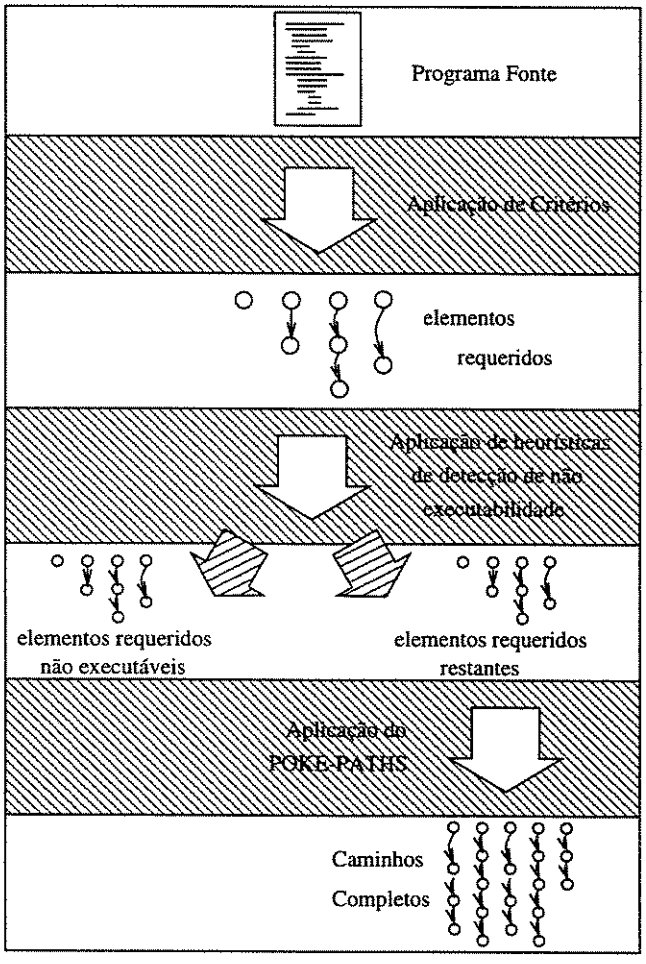

Figura 4.1: Contexto da geração de caminhos completos no teste estrutural.

A Figura 4.2 mostra como o módulo Poke-Paths se relaciona com a ferramenta Poke-Tool.

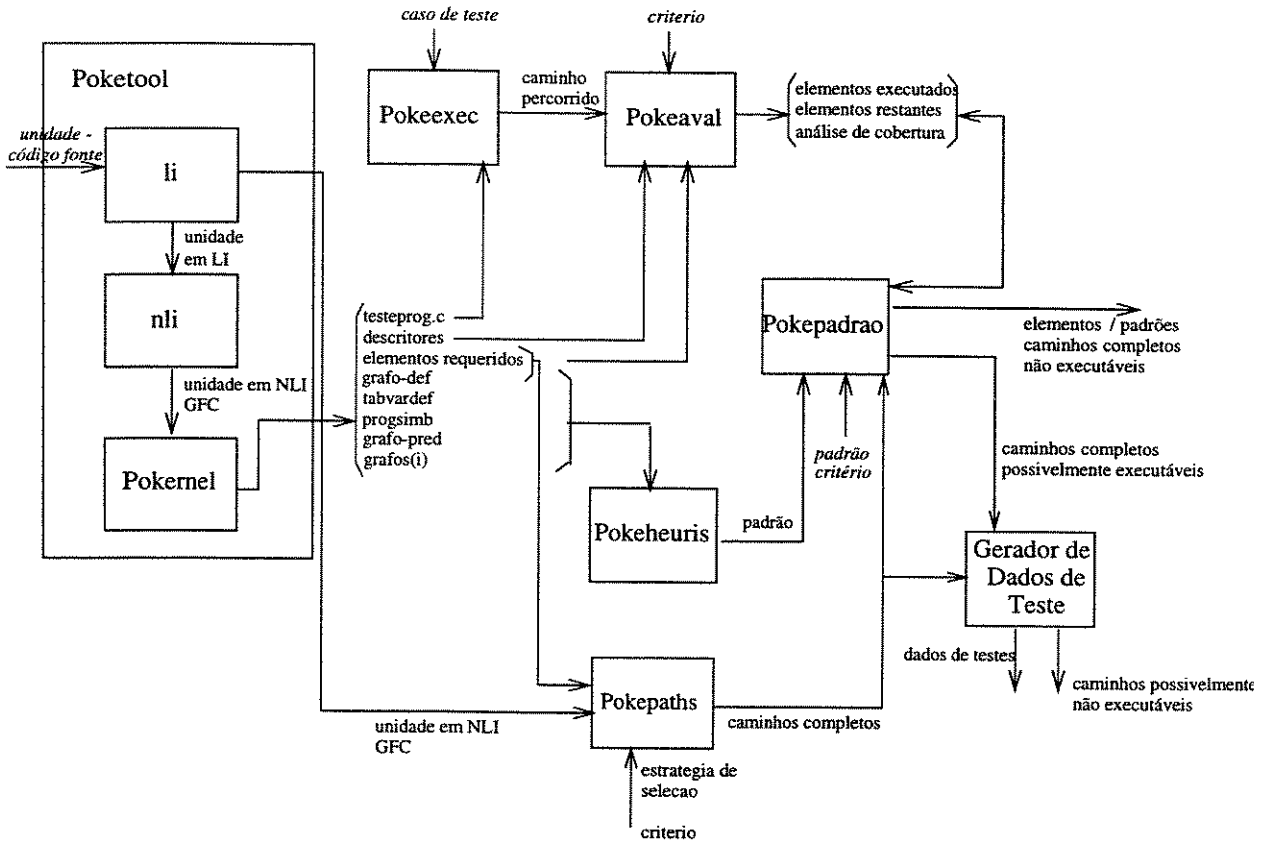

Figura 4.2: Arquitetura da ferramenta Poke-Tool, incluindo Poke-Paths.

4.3 Solução Implementada

O Poke-Paths utilizou-se da estrutura de representação de estratégias, proposta no Capítulo 3. A solução apresentada nesta seção tem como base o par ordenado $S = (p(n), e(\pi))$, onde $p(n)$ é implementado através de um campo *peso* (que contém o valor de $p(n)$), contido no registro *nó* (que refere-se a n), que faz parte da estrutura *grafo*; e $e(\pi)$ implementado através de dois procedimentos: um que gera os k-menores caminhos (associados à seleção dos k-menores caminhos, como definido no Capítulo 3) e outro que gera os k-maiores caminhos (associados à seleção dos k-maiores caminhos, como definido no Capítulo 3)

Com base na definição do problema, e considerando o contexto em que este módulo está inserido, desenvolveu-se uma solução que está estruturada como apresentada no Diagrama de Fluxo de Dados (DFD) [45], na Figura 4.3. Esta figura é simplificadamente comentada abaixo.

O módulo desenvolvido recebe como entrada opções do usuário e dados de arquivos gera-

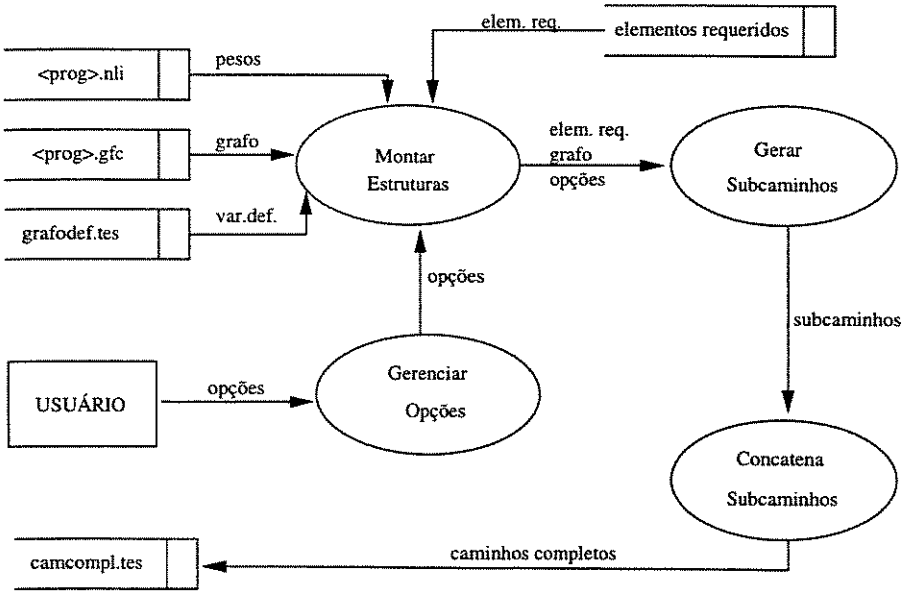

Figura 4.3: DFD representando a solução de implementação.

dos pela ferramenta Poke-Tool. De acordo com as opções que são entradas através da bolha "Gerenciar Opções", realizam-se a leitura dos arquivos, na bolha "Montar Estruturas": <programa>.gfc, grafodef.tes, arquivos de elementos requeridos e quaisquer arquivos que contenham informações de estratégias (na Figura 4.3, como exemplo tem-se <prog>.nli). Esta leitura proporciona a obtenção dos dados que preencherão as estruturas *grafo* e *elemento_requerido*.

Após a montagem das estruturas de dados referentes aos principais elementos da solução, é possível determinar os nós iniciais e finais dos subcaminhos a serem gerados. Isto porque, de acordo com o Princípio da Otimalidade [34, 33], apresentado no Capítulo 2, a geração de caminhos completos é dividida em geração de subcaminhos que, concatenados após esta geração, formam os caminhos completos. Explicações mais detalhadas a respeito desta divisão são apresentadas na função MontaElementoRequerido.

Na bolha "Gerar Subcaminhos", tem-se três procedimentos que utilizam algoritmos diferentes para a geração de subcaminhos: o que gera os k-menores subcaminhos, o que gera os k-maiores subcaminhos (estes dois implementando a função $e(\pi)$ do par ordenado S) e o que gera subcaminhos aleatórios.

Após a geração dos conjuntos de subcaminhos, estes são concatenados ordenadamente, formando caminhos completos. Esta operação é representada pela bolha "Concatenar Subcami-

nhos".

Como resultado do processamento, o módulo Poke-Paths gera um arquivo de caminhos completos que cobrem os elementos requeridos passados como opção de entrada.

As próximas seções descrevem a arquitetura do módulo Poke-Paths, apresentando em maiores detalhes as funções que implementam a estrutura de representação de estratégias. A seção seguinte descreve a implementação específica da estratégia menor número de predicados.

4.3.1 Arquitetura

A implementação da solução baseia-se no paradigma estruturado de desenvolvimento [45], utilizando-se a linguagem de programação C. O particionamento da solução [45] é apresentado na Figura 4.4.

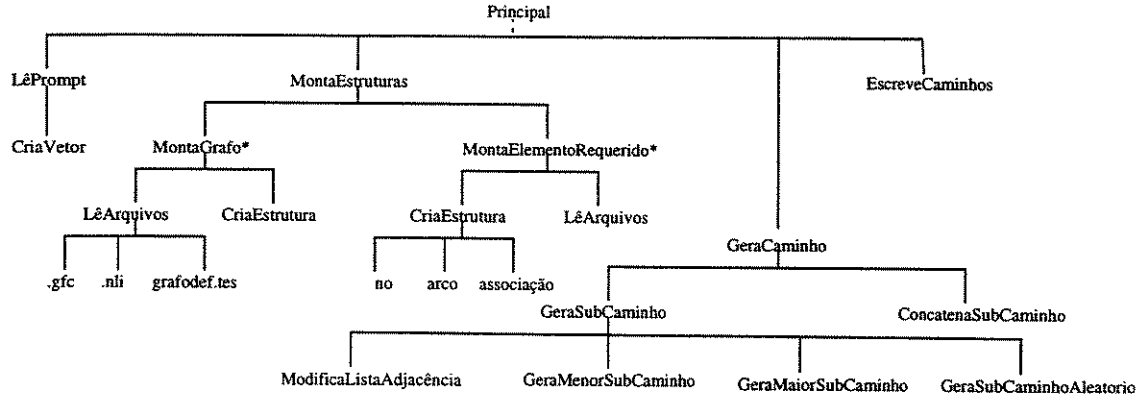

Figura 4.4: Particionamento da Solução.

A partir de um módulo principal, as funções LêPrompt, MontaEstrutura, GeraCaminho e EscreveCaminhos são chamadas. As principais funções são descritas a seguir.

LêPrompt Esta função recebe como entrada os dados digitados pelo usuário na linha de comando e põe em uma estrutura de dados do tipo *vetor*. A seguir, é apresentado o formato de entrada que o usuário deve utilizar para acessar o módulo Poke-Paths [1]:

pokepaths -d<função> -[nos | arcs | pu | pdu] -[all | l<lista> | i'<x> to <y> '|u'<elemento> '] -e<estratégia> [-m] [-c<número>] [-r<número>] -o<arquivo de saída>

[1]Parâmetros entre [ ] são opcionais, | indica "ou exclusivo", palavras entre <> e os símbolos <> devem ser substituídos pelo indicado na explicação do item.

onde:

<função>: nome da função em teste a ter seus caminhos gerados;

-nos: elementos requeridos pelo critério todos-nós;

-arcs: elementos requeridos pelo critério todos-ramos;

-pu: elementos requeridos pelo critério todos-potenciais-usos;

-pdu: elementos requeridos pelo critério todos-potenciais-du-caminhos;

-all: gerar caminhos para todos os elementos requeridos pelo critério e contidos no arquivo de elementos requeridos gerado pela Poke-Tool;

-l<lista>: gerar caminho somente para os elementos referenciados no arquivo <lista>. O conteúdo deste arquivo é uma lista de números ordenados, que representa os elementos no arquivo de elementos requeridos, gerado pela Poke-Tool. O formato deste arquivo é apresentado no Apêndice A;

-i'<x> to <y>': gerar caminho somente para elementos que estão no intervalo x-y de números inteiros. Estes números representam a ordem dos elementos no arquivo de elementos requeridos gerado pela Poke-Tool;

-u'<elemento>': gerar caminhos somente para o <elemento>, fornecido via string;

-e<estratégia>: <estratégia> é um número inteiro indicando a estratégia de geração de caminhos, podendo ser atualmente:

1. número de predicados
2. aleatória
3. qualquer estratégia[2]

-m: indica qual instância de $e(\pi)$ deve ser usada, ou seja, se deve-se gerar os caminhos com *menores* ou *maiores* pesos. O *default* é encontrar os caminhos com *menores* pesos.

[2]Esta opção permite que se entre com os pesos associados aos nós manualmente, possibilitando a utilização de quaisquer estratégias apresentadas no Capítulo 3.

-c<número>: <número> é um inteiro indicando o número de caminhos completos que devem ser gerados. Default = 1. Caso se deseje todos os caminhos possíveis, de acordo com o número de ciclos e tipo de elemento requerido, <número> deve ser igual a 0 (zero);

-r<número>: <número> é um inteiro indicando o número de vezes que é permitida a entrada em um ciclo. Default = 1;

-o<arquivo>: opção que permite definir o arquivo de saída. O nome de saída padrão é ' cp<critério><estratégia>.tes';

Mais detalhes a respeito das opções, seus objetivos e suas implicações são detalhadas no Apêndice A.

MontaEstruturas O objetivo geral desta função é preparar as principais estruturas de dados para serem utilizadas no processo GeraCaminhos, que é o de geração de caminhos propriamente dito.

Um procedimento inicial de leitura de arquivos é efetuado, onde são realizadas: 1) a leitura dos arquivos <programa>.gfc e grafodef.tes para a criação do grafo; 2) a leitura de arquivos referentes à associação de pesos aos nós; e 3) a leitura de um dos arquivos referentes aos elementos requeridos. Os arquivos citados nestes itens são oriundos da ferramenta Poke-Tool.

O item 2 do parágrafo anterior é um dos meios de instanciar a função $p(n)$ do par ordenado S, definido no Capítulo 3, requerido na seção anterior e implementado neste módulo. Para a estratégia menor número de predicados faz-se a leitura do arquivo <programa>.nli.

Duas estruturas principais são criadas e recebem valores nesta função: *grafo* (em MontaGrafo), que reflete o grafo def, e *elemento_requerido* (em MontaElementoRequerido), que reflete o elemento requerido por um critério. Estas estruturas são consideradas os principais elementos a serem manuseados nesta solução. Considerando o grafo def, isto se deve ao fato de nela estarem refletidas as informações a respeito dos nós, como peso, lista de adjacência de nós e definição de variáveis. A associação de pesos aos nós, como já mencionado, implementa a função $p(n)$ de S. A lista de adjacências, que contém os

vértices adjacentes a cada nó, é base para os algoritmos gerarem caminhos. Através da lista de variáveis definidas faz-se a análise de caminhos livres de definição.

A estrutura de elementos requeridos tem sua importância na definição dos nós inicial e final dos subcaminhos a serem gerados, garantindo-se a geração de caminhos que cobrem os elementos requeridos pelo critério desejado.

MontaGrafo Utilizando os analisadores léxico *Lex* e sintático *Yacc* [27] para a leitura e interpretação dos arquivos, foi possível a atribuição de dados à estrutura *grafo* de maneira fácil e eficiente.

De acordo com a definição do par ordenado $S = (p(n), e(\pi))$ apresentada no Capítulo 3, Seção 3.1, para as estratégias de seleção de caminhos descritas, associam-se pesos aos nós, como definido pela função $p(n)$ de S. Para a implementação desta definição, projetou-se o *nó* da estrutura *grafo*(grafo def) como um registro, que possui como um dos campos a estrutura *peso*. Isto permite que quaisquer das estratégias definidas no Capítulo 3 sejam facilmente implementadas.

A estrutura grafo é uma lista de nós. A estrutura nó possui o seguinte formato:

```
struct no {
  int id;          /* numero do no */
  double peso;     /* representa a funcao p(n) */
  t_no *listaadj ;   /* lista de adjacencia de nos vizinhos */
  t_variaveis *listadefinicao;  /* lista de definicao de variaveis*/
};
```

Para cada estratégia definida no Capítulo 3, é possível instanciá-la para concretizar sua implementação neste módulo. Esta instanciação se dá com a atribuição de pesos aos nós, seja por meio de entrada manual pelo usuário (implementada através da opção -e3) ou seja por meio da leitura de arquivos que contenham informações relevantes à estratégia. Atualmente, somente a estratégia menor número de predicados é implementada com instanciação através de leitura de arquivo (mais detalhes podem ser encontrados na Seção

4.3.2).

MontaElementoRequerido O objetivo desta função é preparar a estrutura *elemento_requerido* para ser utilizada na função GeraSubCaminho (de GeraCaminho). "Preparar" significa criar a estrutura *elemento_requerido* e atribuir valores a ela. Estes valores são lidos dos arquivos de elementos requeridos e atribuídos à estrutura utilizando as ferramentas *Lex* e *Yacc*. A estrutura em questão é uma lista de vetores de duas posições: nó inicial e nó final, que guardam respectivamente os nós inicial e final de cada subcaminho a ser gerado.

Dependendo do critério escolhido, passado através do vetor de opções, tem-se um arquivo de elemento requerido a ser lido e um tipo de elemento a ser criado, podendo ser nó, arco, potencial associação ou potencial du-caminho. A lista a seguir indica, para cada opção de entrada referente ao critério de teste, o arquivo oriundo da Poke-Tool que contém os elementos requeridos e o tipo de elementos a ser tratado:

-nos: o arquivo é nos_grf.tes e o elemento requerido é do tipo nó;

-arcs: o arquivo é arcprim.tes e o elemento requerido é do tipo arco;

-pu: o arquivo é puassoc.tes e o elemento requerido é do tipo potencial associação;

-pdu: o arquivo é pdupaths.tes e o elemento requerido é do tipo potencial du-caminho;

Segundo Yates e Malevris, como visto no Capítulo 2 e Martins, Pascoal e Santos [34], o Princípio da Otimalidade afirma que há um menor caminho formado por menores subcaminhos. No trabalho de Martins *et al.* é provado que o k-ésimo menor caminho é formado pelos j-ésimos menores caminhos, para j <= k. Esta formação é dada pela operação concatenação, definida no Capítulo 2.

A decomposição de um elemento requerido para a geração de conjuntos de subcaminhos é mostrada com o auxílio da Figura 4.5.

Considerando a Figura 4.5(a) para a geração de um caminho completo que cubra um *nó* i, geram-se dois conjuntos de subcaminhos A, cujos subcaminhos possuem nó inicial e (e_A) e nó final i (s_A); e B, cujos subcaminhos possuem nó inicial i (e_B) e final s (s_B).

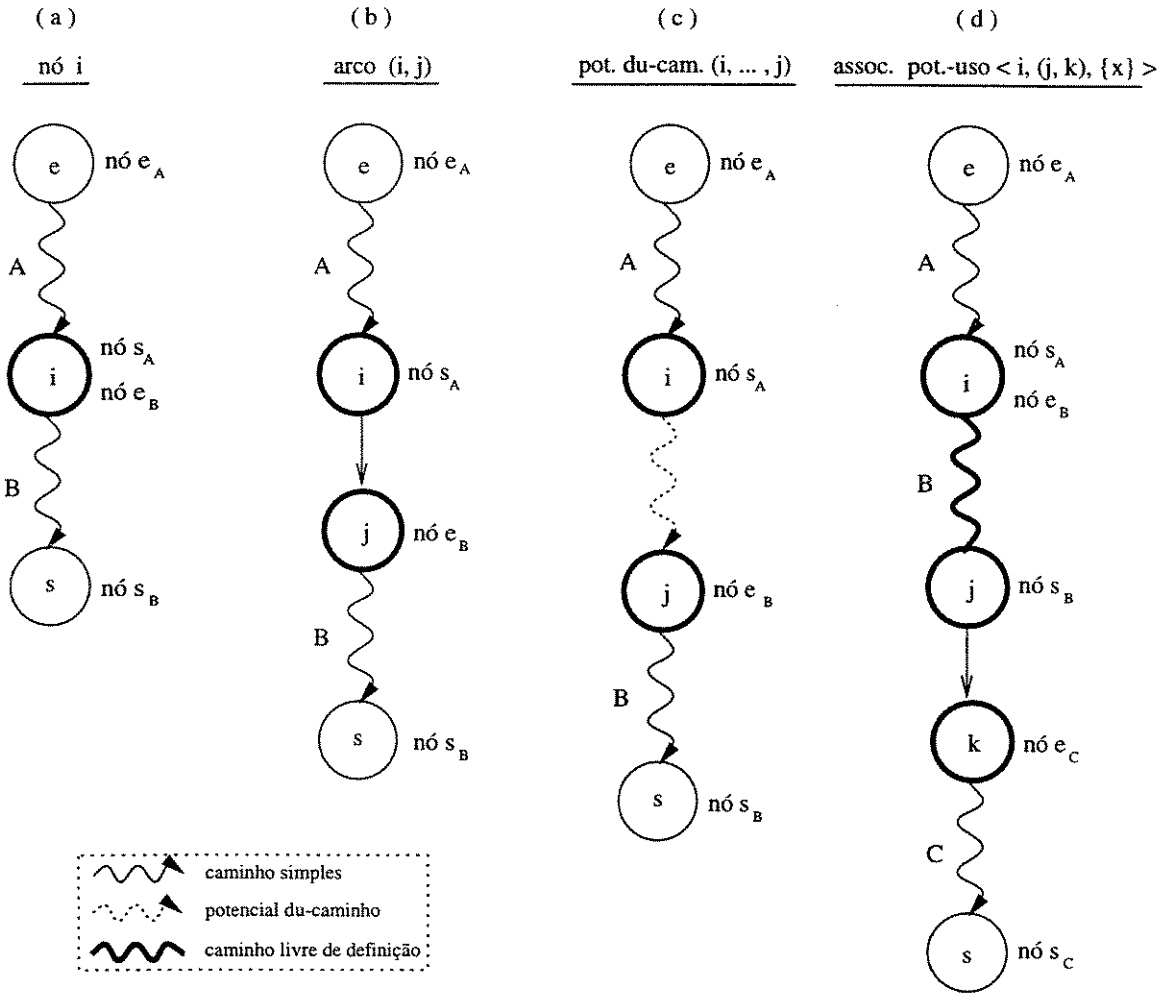

Figura 4.5: Divisão em subgrafos para a geração de subcaminhos.

Sendo o elemento requerido um *arco* (i, j), deve-se gerar dois conjuntos de subcaminhos, onde todos os caminhos pertencentes ao primeiro conjunto A devem possuir como nó inicial e (e_A) e como nó final i (s_A); o conjunto B deve possuir caminhos que iniciem no nó j (e_B) e terminem no nó s (s_B), como apresentado na Figura 4.5(b).

Na Figura 4.5(c), para um *potencial du-caminho* $(i, ..., j)$, geram-se dois conjuntos de caminhos onde seus componentes tem a obrigatoriedade de: no conjunto A, possuir nó inicial e (e_A) e nó final i (s_A); no conjunto B, possuir nó inicial j (e_B) e nó final s (s_B).

Caso o critério escolhido pelo testador seja todos-potenciais-usos, requere-se uma potencial associação como mostrado na Figura 4.5(d). Para cobrir uma *potencial associação* $< i, (j, k), \{x\} >$, requere-se a geração de três conjuntos de subcaminhos: A, B e C. Todos

os caminhos pertencentes ao conjunto A devem possuir como nó inicial e (e_A) e nó final i (s_A); os caminhos pertencentes ao conjunto B devem possuir como nó inicial i (e_B) e nó final j (s_B), sendo que todos estes caminhos devem ser livres de definição com relação à variável x; os caminhos pertencentes ao conjunto C devem possuir nó inicial k (e_C) e nó final s (s_C).

GeraCaminho As atividades de geração de subcaminhos, e a concatenação destes, são realizadas nesta função, em dois procedimentos: GeraSubCaminho e ConcatenaSubCaminhos. A função GeraCaminho recebe como parâmetros a estrutura *elemento_requerido*, ou seja, lista que representa um elemento requerido (montada como apresentado na função MontaElementoRequerido) e a estrutura *grafo*.

GeraSubCaminho Esta função permite três tipos de geração de caminhos: a aleatória e as que selecionam os k-menores e os k-maiores caminhos, respectivamente. Permite ainda que sejam gerados dois tipos de caminhos para cada tipo de geração: caminho simples e caminho livre de definição.

Quando um subcaminho livre de definição em relação a uma ou mais veriáveis deve ser gerado, antes da chamada às funções de geração propriamente ditas (GeraMaiorSubCaminho, GeraMenorSubcaminho, GeraSubCaminhoAleatório), a função ModificaListaAdjacência realiza operações sobre a estrutura grafo, eliminando da lista de adjacência nós adjacentes que possuem no mínimo uma definição da(s) variável(is) em questão.

Para a geração dos k-ésimos menores e maiores subcaminhos foram utilizados dois algoritmos que são descritos em funções a seguir. Chong *et al.* [14] apresentam um algoritmo para os k-ésimos menores caminhos, que foi implementado na função GeraMenoresCaminhos. Este algoritmo foi modificado para encontrar os k-ésimos maiores caminhos e foi implementado na função GeraMaioresCaminhos.

ModificaListaAdjacência Como apresentado no Capítulo 2 e na Figura 4.5, para uma potencial associação <i,(j,k),{x}>, todos os subcaminhos a serem gerados de i para j, devem ser livres de definição c.r.a x. Para implementar tal restrição, a estrutura *lista_de_adjacên-*

cia, contida em cada estrutura *nó* do *grafo*, é modificada de modo que apenas permanece nesta lista os nós que não contêm definições de x, seguindo a definição dada no Capítulo 2.

Se após esta função ter sido executada não houver nenhum nó na estrutura *lista_de_adjacência* de um dado nó que não seja o final, não existirá um caminho livre de definição entre os nós inicial e final do que seria um subcaminho a ser gerado. A execução desta função pode gerar um grafo desconexo o que não invalida este módulo ou a utilização dos algoritmos, pois os algoritmos utilizados operam sobre um grafo desconexo.

Após a *lista_de_adjacência* da estrutura *grafo* ser modificada, este é passado como parâmetro às funções GeraMaiorSubCaminho, GeraMenorSubCaminho ou GeraSubCaminho-Aleatório, mostradas na Figura 4.4, para que seja utilizado como base para a geração de subcaminhos.

GeraMenorSubCaminho Na implementação desta função, utilizou-se um algoritmo proposto por Chong *et al.* [14], que se baseia no algoritmo de Djikstra para encontrar o menor caminho entre um vértice (neste texto, nó) e todos os outros vértices. O algoritmo de Chong *et al.* encontra o k-menor caminho a partir de um nó inicial até um nó final e pode ser estudado em detalhes, inclusive com exemplos, em [14] .

Este algoritmo possui a restrição de não permitir pesos negativos. Estes números negativos tornam a utilização deste algoritmo inviável quando o grafo é cíclico, pois um k-menor caminho gerado nestas condições é infinito.

GeraMaiorSubCaminho O problema de construção dos k-maiores caminhos em um grafo com ciclos e pesos positivos, é NP-Completo [18], como o problema da geração dos k-menores caminhos com pesos negativos. Em face disto, para gerar os k-maiores caminhos, adaptou-se o algoritmo de Chong *et al.* restringindo o número de ciclos permitidos em um caminho, tornando o grafo acíclico. Nesta implementação, este número é configurado pelo usuário através da opção -r<número>.

Esta decisão de limitação de ciclos para a geração de caminhos é inspirada no trabalho

de Howden [24], que sugeriu a restrição do número de vezes que um ciclo é executado como um critério mais rigoroso que todos-ramos e menos restritivo e dispendioso do que o critério todos-caminhos.

GeraSubCaminhoAleatório A geração de subcaminhos aleatórios não utiliza nenhum tipo de estratégia de seleção de caminhos ou estrutura de representação de estratégias.

A implementação deste algoritmo utiliza as listas de adjacências associadas aos nós do *grafo*. Seleciona-se aleatoriamente um nó da lista de adjacência, para incluir como próximo nó no caminho que está sendo gerado. A partir da lista de adjacência deste nó que acabou de ser incluído, realiza-se a seleção aleatória de outro nó, que também é incluído no caminho, e assim por diante até que um nó incluído no caminho seja um nó que não possui lista de adjacência.

ConcatenaCaminhos Esta função somente concatena os subcaminhos gerados para cada elemento requerido. Esta atividade é feita utilizando-se a operação concatenação definida no Capítulo 2. Considerando as Figuras 4.5(a) e 4.5(b), realiza-se o produto cartesiano da concatenação dos caminhos ordenados contidos no conjunto A com os caminhos ordenados contidos no conjunto B, e, para potenciais associações (Figura 4.5(d)), ainda com os caminhos ordenados contidos no conjunto C. Com relação à Figura 4.5(c), concatenam-se os caminhos ordenados contidos no conjunto A, com o próprio potencial du-caminho e com os caminhos ordenados contidos no conjunto B.

EscreveCaminhos Esta função escreve os caminhos completos gerados, para cada elemento requerido, no arquivo de saída do Poke-Paths. Um exemplo deste arquivo é apresentado no Apêndice C.

4.3.2 Estratégia Menor Número de Predicados

A estratégia de seleção de caminhos menor número de predicados foi implementada no módulo Poke-Paths como um mecanismo de diminuição do número de caminhos não executáveis selecionados.

Como apresentado na seção passada, utiliza-se o campo *peso* no registro *nó* para implementar $p(n)$ e dois algoritmos que encontram os k-ésimos menores e maiores caminhos para implementar $e(\pi)$. Isto quer dizer que, para implementar a estratégia menor número de predicados, bastou-se realizar a leitura do arquivo <programa>.nli para a obtenção automática dos pesos ($p(n)$) que devem ser associados a cada nó (n) e sua respectiva atribuição. O restante do módulo funciona como já descrito anteriormente.

Como a seleção dos caminhos com menor peso já está implementada (através dos algoritmos de k-ésimos menores e maiores caminhos), basta ao usuário escolher se deseja os k-ésimos caminhos com menor ou maior número de predicados. Nenhuma estratégia maior número de predicados foi sugerida no Capítulo 3, apesar da implementação deste módulo permitir a escolha da atribuição de pesos pelo número de predicados e o algoritmo k-maiores caminhos na mesma execução. No entanto, no experimento apresentado no Capítulo 5 geram-se caminhos utilizando a estratégia "maior número de predicados" como um meio de comparar os dados desta estratégia com os dados encontrados aplicando-se a estratégia menor número de predicados. Maiores detalhes são apresentados no Capítulo 5.

4.3.3 Restrições

Para a solução adotada no módulo Poke-Paths, algumas considerações/limitações devem ser colocadas:

- O modelo de fluxo de controle e fluxo de dados é o mesmo adotado pela Poke-Tool;
- A cada execução, o módulo Poke-Paths deve gerar caminhos completos por um único critério. Portanto, os elementos requeridos devem ser de apenas um tipo: apenas nós, arcos, associações ou potencial du-caminhos a cada execução;
- Os caminhos a serem gerados não possuem a obrigatoriedade de serem executáveis;
- Considerando o estado atual de implementação deste módulo, podem ser utilizadas somente estratégias de seleção de caminhos que se enquadrem na estrutura de representação de estratégias $S = (p(n), e(\pi))$;

- Na geração de caminhos completos para rotinas que possuam iterações, é limitado o número de execuções possíveis de um laço;

- De acordo com o projeto de implementação, o módulo Poke-Paths recebe como entrada arquivos gerados pela ferramenta Poke-Tool. Isto implica numa dependência atual do módulo em relação à Poke-Tool. Consequentemente, não é possível gerar caminhos completos sem que os arquivos de entrada tenham sido gerados pela ferramenta Poke-Tool, ou estejam no formato estipulado por esta ferramenta, e que as unidades estejam implementadas nas linguagens de programação que a Poke-Tool aceita.

4.4 Um Procedimento de Teste Usando o Módulo Poke-Paths

Um possível procedimento de teste, proposto por [9] e adaptado ao módulo Poke-Paths, utilizando a ferramenta Poke-Tool baseia-se nos seguintes passos:

1. Utilização do programa Poke-Tool na realização da análise estática do código fonte, na geração dos elementos requeridos para os critérios apoiados pela ferramenta e no programa modificado para teste;

2. Seleção de casos de teste *ad hoc* baseada nos elementos requeridos pelos critérios apoiados, ou por outro critério de seleção, como o aleatório ou algum funcional (caixa preta);

3. Execução do programa com os casos de teste escolhidos utilizando-se o Pokeexec. São armazenados os casos de teste - dados de entrada e saídas respectivas - e os caminhos executados nas unidades de teste;

4. Avaliação da cobertura atingida para o critério escolhido utilizando-se o módulo Pokeaval. São fornecidos os elementos requeridos ainda não executados e os já executados pelos casos de teste;

5. Complementação do conjunto de casos de teste para a satisfação do critério utilizado. Para isto deve-se realizar as atividades:

- Aplicação, através do módulo Pokeheuris, de heurísticas de determinação de não executabilidade de elementos requeridos não executados até este momento. Este procedimento possibilita a identificação e tratamento automatizados de elementos não executáveis para os quais a heurística é aplicável;

- Análise dos elementos requeridos ainda não executados (se ainda existirem) e identificação manual dos não executáveis. Inserção dos padrões associados a estes elementos no arquivo <função>.padrão através do Pokepadrão. Consegue-se assim a manutenção sistemática dos padrões de não executabilidade identificados para o programa, eliminação de elementos que contenham o padrão do arquivo de elementos não executados e reavaliação da cobertura atingida.

- Elementos requeridos não executados e que não foram detectados como não executáveis têm seus caminhos completos gerados pelo módulo Poke-Paths. Pode-se gerar caminhos com maior probabilidade de serem executáveis, através da estratégia menor número de predicados;

- Utilização do Pokepadrão novamente, que identifica, entre os caminhos completos, aqueles que são não executáveis, de acordo com a base de padrões de não executabilidade existente no arquivo <função>.padrão.

- Geração de dados de teste, utilizando o módulo de geração de dados de teste de Bueno [7] a partir de caminhos completos executáveis e realimentação da base de não executabilidade no arquivo <função>.padrão, para os caminhos considerados não executáveis;

- Execução dos dados de teste encontrados na etapa anterior;

- Avaliação da cobertura atingida para o critério escolhido, utilizando-se o módulo Pokeaval;

6. Avaliação dos casos de teste encontrados para a satisfação dos critérios, através de um oráculo.

4.5 Considerações Finais

O módulo Poke-Paths implementa a estrutura de representação de estratégias proposta no Capítulo 3 e pode ser instanciado para quaisquer estratégias de seleção de caminhos apresentadas.

Para a implementação descrita neste capítulo, utilizou-se os elementos requeridos pelos critérios todos-nós, todos-arcos, todos-potenciais-du-caminhos e todos-potenciais-usos apoiados pela ferramenta Poke-Tool.

A utilização de uma estrutura diferente da *grafo(i)*[3], implementada pela ferramenta Poke-Tool, permite que diferentes critérios de teste estrutural tenham caminhos completos gerados para seus elementos requeridos. Este módulo permite a extensão para pelo menos os critérios baseados na Família de Critérios de Fluxo de Dados (DFCF), bem como os critérios estruturais baseados no fluxo de controle. Quaisquer outros critérios que possam ter seus elementos requeridos reduzidos a nós, arcos, associações definição-uso de variáveis ou subcaminhos podem ter seus caminhos completos gerados pelo módulo Poke-Paths. A utilização de outros critérios torna este módulo um excelente protótipo de geração de caminhos, que possibilita a realização de diferentes experimentos que necessitem desta potencialidade.

Este módulo implementa a geração de dois tipos de caminhos: aqueles escolhidos por uma estratégia de seleção de caminhos para quaisquer definições de p(n) e $e(\pi)$ descritas no Capítulo 3 e caminhos gerados aleatoriamente, que não se utilizam de nenhuma estratégia de seleção de caminhos.

Utilizou-se o Princípio da Otimalidade, ou seja, a divisão da geração de caminhos em geração de subcaminhos que são posteriormente concatenados, sendo que as estratégias de seleção de caminhos são válidas para a geração de cada subcaminho. O projeto do módulo Poke-Paths, devido à generalidade de suas estruturas de dados, permite a incorporação de outros algoritmos para encontrar menores e maiores caminhos.

[3]Estrutura de dados utilizada na ferramenta Poke-Tool, que é obtida a partir do grafo def e fornece todos os caminhos livres de definição com relação a qualquer variável definida em i para todo nó e todo arco alcançável a partir do nó i. A proposição é a de se construir um único grafo(i) para cada nó i tal que $defg(i)$ é diferente de vazio, obtendo-se por construção uma minimização de caminhos e associações requeridos. Na realidade, os grafo(i) incluem somente os potenciais-du-caminhos a partir do nó i [11, 28, 54].

Segundo Bertolino e Marré [3], um programa de geração de caminhos deve apresentar mecanismos para a diminuição do número de caminhos não executáveis gerados. Para satisfazer esta restrição os seguintes mecanismos estão disponíveis:

- Inclusão de elementos requeridos executáveis, como entrada. É possível a utilização do módulo Pokeheuris, antes da execução de Poke-Paths, para a provável identificação e exclusão de alguns elementos não executáveis, anteriormente à geração de caminhos completos;
- Geração de caminhos utilizando a estratégia de seleção de caminhos proposta por Malevris, que requer que caminhos com menor número de predicados sejam selecionados, aumentando a probabilidade de selecionar caminhos executáveis;
- Utilização do módulo Pokepadrão após a geração dos caminhos completos, de modo a identificar padrões de não executabilidade em um conjunto de caminhos completos gerados automaticamente.

O módulo Poke-Paths foi instanciado com a estratégia menor número de predicados com o objetivo de realizar uma avaliação dessa estratégia com relação ao número de caminhos executáveis selecionados. Resultados dessa avaliação são descritos no próximo capítulo.

Capítulo 5

Aplicação da Estratégia Menor Número de Predicados

Neste capítulo, apresentam-se duas aplicações do módulo Poke-Paths em conjunto com a ferramenta Poke-Tool. Estas aplicações têm como objetivo validar o módulo implementado e definir alguns procedimentos para a sua utilização, além da observação do comportamento da estratégia menor número de predicados com relação ao número de caminhos executáveis selecionados e à eficácia dos caminhos em termos do número de erros revelados.

Estas aplicações comparam a estratégia menor número de predicados (ou menor n.p.) com a estratégia oposta, "maior número de predicados" (ou maior n.p.), e a seleção aleatória de caminhos. Define-se a estratégia "maior número de predicados", utilizando a estrutura de representação para estratégias apresentada no Capítulo 3, como um par ordenado $S = (p(n), e(\pi))$, onde a função $p(n)$ é o número de predicados do bloco de comando correspondente ao nó n_i e $e(\pi)$ é uma função que seleciona o caminho com maior peso total (P_π).

5.1 Número de Caminhos Executáveis

A primeira aplicação realizada avaliou as estratégias menor n.p., maior n.p. e aleatória quanto ao número de caminhos executáveis e utilizou os elementos requeridos pelos critérios de teste todos-ramos [17] e todos-potenciais-usos [28]. O objetivo desta aplicação é observar o número de caminhos executáveis selecionados para estas estratégias para cobrir os elementos requeridos por estes critérios.

5.1.1 Metodologia

Para este experimento definiu-se a utilização de quatro programas extraídos de *benchmarks* conhecidos e utilizados em outros trabalhos de teste de software: *cal* e *comm* [60, 61, 55](programas do Unix), *entab* e *expand* [59, 58, 28, 54, 55]. Dentre os programas utilizados, os dois últimos possuem uma baixa complexidade e os dois primeiros complexidade média, segundo as métricas LOC, Contagem de Tokens, Complexidade Ciclomática e Nível de Aninhamento [12, 45]. As medições realizadas são apresentadas no Apêndice B.

Algumas características interessantes dos programas são destacadas abaixo de modo a facilitar a análise dos resultados. O Apêndice B apresenta mais detalhes.

- Programa *cal*: a rotina *pstr* possui dois laços "while" seqüenciais e dependentes.
- Programa *comm*: a rotina *main* apresenta um laço de aninhamento no nível 0 (ver Tabela B.2 do Apêndice B) "while(1)", dentro deste laço, num aninhamento de nível 4 , tem-se o comando de saída deste laço ("exit(0)").

A aplicação deste experimento consistiu de duas atividades principais: a geração do conjunto de elementos requeridos e a geração de dados de teste.

Abaixo estão descritas as principais etapas que foram seguidas. Todas as etapas a seguir foram realizadas para todos os programas no contexto do teste de unidade.

1. geração de elementos requeridos pelo critério baseado em fluxo de controle todos-ramos e pelo critério baseado em fluxo de dados todos-potenciais-usos;
2. execução de um conjunto inicial de dados de teste e avaliação da cobertura dos elementos requeridos gerados na etapa anterior;
3. geração dos cinco melhores caminhos[1], através do módulo Poke-Paths, para os elementos executados selecionados para este experimento[2], com exemplos no Apêndice B, de acordo com as estratégias de seleção de caminhos:

[1] o número de caminhos foi limitado para evitar uma explosão combinatorial.
[2] o experimento não foi realizado com todos os elementos executáveis

- menor n.p.;
- maior n.p.;
- aleatória.

Para casos onde há empate no número de predicados, na seleção de caminhos usando as estratégias menor n.p. e maior n.p., foi adotado como critério de desempate a existência do menor número de nós nos caminhos, pois a idéia é gerar caminhos mais simples para facilitar a automação da geração de caminhos.

Na seleção dos caminhos através das estratégias maior n.p. e aleatória, é necessário impor um limite de entrada em ciclos, como apresentado no Capítulo 4. Isto porque na presença de ciclos, sem limite de entradas no ciclo, e pesos positivos o processo de geração de caminhos será infinito. Este limite foi estabelecido em permitir 2 entradas no ciclo.

4. determinação e contagem da executabilidade dos caminhos gerados. Alguns critérios foram adotados a respeito da determinação de não executabilidade:

   (a) Como o contexto deste experimento é o teste de unidade, considerou-se cada unidade analisada como livre de contexto, assim como sugerido por Frankl e Weyuker [17]. Não foi considerada a não executabilidade imposta por variáveis globais e por valores passados como parâmetros em chamadas e retornos de procedimentos.

   (b) A determinação de executabilidade em caminhos que contenham laços foi avaliada segundo os dois critérios a seguir:

      i. Determinação de executabilidade sem diferenciação para qualquer tipo de laço.

      ii. Determinação de executabilidade diferenciada para laços onde é possível determinar estaticamente o número de execuções do mesmo: qualquer caminho que entra ao menos uma vez em um laço deste tipo é considerado executável; e todo caminho que não entra é considerado não executável.

      Um exemplo apresentado na linguagem Fortran, por Hedley e Hennell [21] é ilustrativo:

```
      DO 20 J = 1,10
          ...
 20 CONTINUE
```

ou

```
      N=2
      DO 40 J = 1,N
          ...
 40 CONTINUE
```

Considerando o primeiro exemplo dado, esta determinação implica na prática que um caminho gerado entrando apenas uma vez no laço (e não 10, como deveria ser) é considerado executável, pois devido à limitação do número de laços deste experimento, nunca será gerado um caminho realmente executável. Este critério foi adotado pois é possível a um algoritmo determinar estaticamente a existência deste tipo de laço e fazer os tratamentos[3] devidos.

Segundo Hedley e Hennell [21] este tipo de não executabilidade acontece em aproximadamente 53% dos casos de não executabilidade.

A escolha deste critério é uma sugestão de Yates e Malevris [62, 32], como apresentado na Seção 2.4; Bertolino e Marré [3], que sugerem tratamento para laços que devam ser iterados seqüencialmente o mesmo número de vezes; e tem precedentes no trabalho de Nejmeh [40], que considera as entradas em qualquer tipo de laço como sendo uma única entrada.

[3]Estes tratamentos poderiam incluir a geração manual de dados de teste, considerando a verdadeira quantidade de entradas no laço, ou ainda, através de uma extensão do módulo Poke-Paths, a detecção automática deste tipo de laço e a conseqüente geração de caminhos com entradas nestes laços acima dos estabelecidos pelo limite de entrada nos laços (opção -r).

No Apêndice C, é possível encontrar resultados das medições utilizando ambos os critérios definidos. Nas tabelas apresentadas neste capítulo, o critério adotado é (b.ii). Esta escolha afetou somente o resultado da rotina *main* do programa *cal*.

5.1.2 Totalização de Caminhos

A etapa 4 da aplicação do experimento culminou em totalizações, por elemento requerido, do número de caminhos executáveis e não executáveis por estratégia de geração de caminhos, apresentadas no Apêndice C. As tabelas apresentadas nesta seção, tal como a Tabela 5.1, foram compiladas a partir destas totalizações e possuem oito colunas: a primeira refere-se às rotinas de cada programa avaliado; da segunda à sétima colunas são apresentados valores que representam quantidades de caminhos executáveis, "exec", e não executáveis, "Nexec", selecionados segundo as estratégias de seleção de caminhos menor n.p., "Menor", maior n.p., "Maior", e aleatória, "Aleatória". A oitava, e última coluna, "Maior Número de Executáveis", resume, dentre as estratégias, aquela que gerou o maior número de caminhos executáveis. Como o número total de caminhos gerados para cada estratégia é sempre o mesmo, é possível fazer uma comparação direta somente entre os valores de caminhos executáveis apresentados. A linha "total", que aparece em todas as tabelas, refere-se à soma dos caminhos gerados para cada função do programa em questão, ou seja, refere-se à totalização de caminhos gerados para tal programa, e a linha "total %" refere-se à porcentagem de caminhos executáveis selecionados para cada estratégia, considerando o total de caminhos (executáveis e não executáveis) selecionados pela estratégia.

Como um exemplo, tem-se que na Tabela 5.1, referente às totalizações do programa *cal* para o critério todos-potenciais-usos, a função *cal* possui entre os caminhos gerados 234 caminhos executáveis, "exec", e 161 caminhos não executáveis, "Nexec", gerados de acordo com a estratégia menor n.p., "Menor". Entre os caminhos gerados de acordo com a estratégia maior n.p., "Maior", obteve-se 0 caminhos executáveis e 395 não executáveis enquanto que 87 caminhos executáveis e 308 não executáveis foram gerados aleatoriamente ("Aleatória"). Comparando-se o número de caminhos executáveis gerados para cada estratégia, tem-se que, para a função *cal*, a estratégia menor n.p., "Menor", gerou o maior número de caminhos executáveis, resultado que é apresentado na coluna "Maior Número de Executáveis". A penúltima linha da tabela, "to-

tal", apresenta a somatória dos valores de caminhos gerados para cada função de *cal* e a última linha "total %" apresenta estas totalizações em porcentagem para cada estratégia. Portanto, para o programa *cal*, foram gerados 319 caminhos executáveis (que representam 46,03% dos caminhos selecionados por esta estratégia, como indicado na última linha "total %") e 374 não executáveis (que representam 53,97%), para a estratégia menor n.p., 70 executáveis (10,10% dos caminhos selecionados por esta estratégia) e 623 não executáveis(89,90%) para a estratégia maior n.p. e 157 caminhos executáveis (22,66%) e 536 não executáveis (77,34%) para a geração aleatória. Conclui-se a partir destes dados, que a estratégia que seleciona o maior número de caminhos executáveis gerados para o programa *cal* é a menor n.p..

Os resultados obtidos para o critério todos-potenciais-usos, têm sua totalização apresentada, respectivamente, para os programas *cal*, *comm*, *entab* e *expand*, nas Tabelas 5.1, 5.2, 5.3 e 5.4. A totalização geral dos dados dos programas, para este critério, é apresentada na Tabela 5.9 e um gráfico para a visualização dos números em porcentagem de caminhos executáveis, é apresentado na Figura 5.1.

Os dados obtidos com relação ao critério todos-ramos, têm sua totalização apresentada, respectivamente para os programas *cal*, *comm*, *entab* e *expand*, nas Tabelas 5.5, 5.6, 5.7 e 5.8. A totalização geral dos dados dos programas, para este critério, é apresentada na Tabela 5.10 e um gráfico para a visualização dos números em porcentagem de caminhos executáveis, é apresentado na Figura 5.2.

Tabela 5.1: Total de caminhos - critério todos-potenciais-usos (programa *cal*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
cal	234	161	0	395	87	308	*Menor*
jan1	11	2	11	2	11	2	*Menor Maior Aleatória*
main	54	146	7	193	27	173	*Menor*
pstr	20	65	52	33	32	53	*Maior*
total	319	374	70	623	157	536	*Menor*
total %	46,03	53,97	10,10	89,90	22,66	77,34	*Menor*

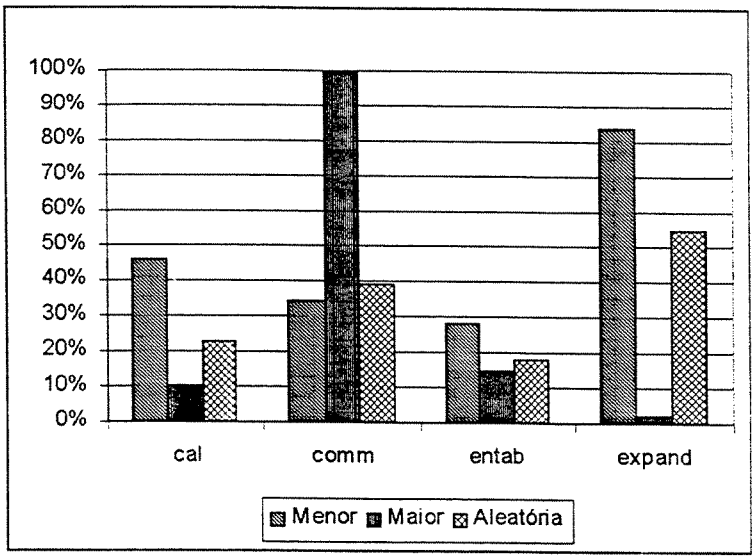

Figura 5.1: Porcentagem de caminhos executáveis - critério todos-potenciais-usos.

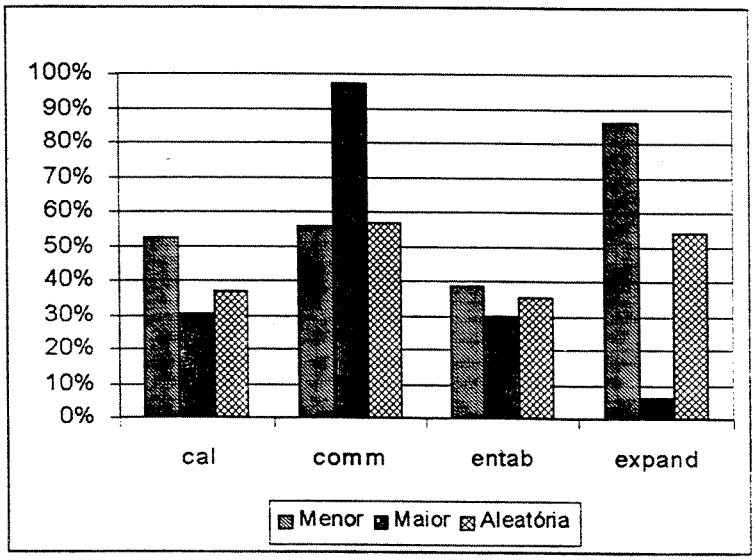

Figura 5.2: Porcentagem de caminhos executáveis - critério todos-ramos.

Tabela 5.2: Total de caminhos - critério todos-potenciais-usos (programa *comm*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
compare	19	0	19	0	19	0	*Menor Maior Aleatória*
copy	8	0	8	0	8	0	*Menor Maior Aleatória*
main	384	865	1244	5	444	805	*Maior*
openfil	2	0	2	0	2	0	*Menor Maior Aleatória*
rd	39	0	39	0	39	0	*Menor Maior Aleatória*
wr	1	0	1	0	1	0	*Menor Maior Aleatória*
total	453	865	1313	5	513	805	*Maior*
total %	34,37	65,63	99,62	0,38	38,92	61,08	*Maior*

5.1.3 Observações por Rotinas

A análise das tabelas apresentadas na seção anterior baseia-se na consideração de qual estratégia gerou o maior número de caminhos executáveis (apresentada na coluna "Maior Número de Executáveis"). Esta seção apresentará análises mais detalhadas dos resultados gerados pelas tabelas apresentadas na seção anterior. As Tabelas 5.11 e 5.12 apresentam a porcentagem que cada estratégia selecionou de caminhos executáveis, considerando o total de caminhos executáveis gerados. Estas tabelas referem-se a porcentagens de caminhos gerados com relação aos critérios todos-potenciais-usos e todos-ramos, respectivamente e possuem cinco colunas cada. A primeira coluna de cada tabela refere-se ao nome das rotinas analisadas, a segunda, terceira e quarta colunas referem-se à porcentagem que cada estratégia ("Menor %", "Maior %" e "Aleatória %", respectivamente) selecionou de caminhos executáveis de um total de caminhos ("Total"), que é apresentado na última coluna. As linhas "comm*" e "total*" apresentam valores percentuais excluídos os valores referentes à função *main* do programa *comm*, como será explicado adiante.

Analisando a Tabela 5.1, percebe-se que o programa *cal* possui 2 rotinas do total de 4, onde a estratégia menor n.p. selecionou o maior número de caminhos executáveis. Existe um empate entre as três estratégias para a rotina *jan1* que, devido à sua simplicidade (ver Tabela B.1, no Apêndice B), possui apenas 13 caminhos gerados para ela, tornando a avaliação independente da estratégia de seleção de caminhos.

Tabela 5.3: Total de caminhos - critério todos-potenciais-usos (programa *entab*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
entab	41	219	0	260	10	250	*Menor*
main	0	0	0	0	0	0	*Menor Maior Aleatória*
settabs	45	1	45	1	45	1	*Menor Maior Aleatória*
tabpos	0	0	0	0	0	0	*Menor Maior Aleatória*
total	86	220	45	261	55	251	*Menor*
total %	28,10	71,90	14,71	85,29	17,97	82,03	*Menor*

Tabela 5.4: Total de caminhos - critério todos-potenciais-usos (programa *expand*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
expand	109	21	3	127	71	59	*Menor*
main	0	0	0	0	0	0	*Menor Maior Aleatória*
total	109	21	3	127	71	59	*Menor*
total %	83,85	16,15	2,31	97,69	54,62	45,38	*Menor*

Tabela 5.5: Total de caminhos - critério todos-ramos (programa *cal*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
cal	40	15	0	55	13	42	*Menor*
jan1	6	2	6	2	6	2	*Menor Maior Aleatória*
main	23	30	22	31	23	30	*Menor Aleatória*
pstr	5	20	15	10	10	15	*Maior*
total	74	67	43	98	52	89	*Menor*
total %	52,48	47,52	30,50	69,50	36,88	63,12	*Menor*

Tabela 5.6: Total de caminhos - critério todos-ramos (programa *comm*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
compare	14	0	14	0	14	0	*Menor Maior Aleatória*
copy	8	0	8	0	8	0	*Menor Maior Aleatória*
main	56	79	130	5	58	77	*Maior*
openfil	3	0	3	0	3	0	*Menor Maior Aleatória*
rd	11	0	11	0	11	0	*Menor Maior Aleatória*
wr	7	0	7	0	7	0	*Menor Maior Aleatória*
total	99	79	173	5	101	77	*Maior*
total %	55,62	44,38	97,19	2,81	56,74	43,26	*Maior*

Tabela 5.7: Total de caminhos - critério todos-ramos (programa *entab*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
entab	6	34	0	40	4	36	*Menor*
main	0	0	0	0	0	0	*Menor Maior Aleatória*
settabs	14	1	15	0	14	1	*Maior*
tabpos	2	0	2	0	2	0	*Menor Maior Aleatória*
total	22	35	17	40	20	37	*Menor*
total %	38,60	61,40	29,82	70,18	35,09	64,91	*Menor*

Tabela 5.8: Total de caminhos - critério todos-ramos (programa *expand*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
expand	30	5	2	33	19	16	*Menor*
main	0	0	0	0	0	0	*Menor Maior Aleatória*
total	30	5	2	33	19	16	*Menor*
total %	85,71	14,29	5,71	94,29	54,29	45,71	*Menor*

A estratégia maior número de predicados, para os critérios todos-potenciais-usos (Tabela 5.1) e todos-ramos (Tabela 5.5), foi a que selecionou maior número de caminhos executáveis para o módulo *pstr*. Isto se deve ao fato desta função possuir um tipo de não executabilidade de dois laços seqüenciais com predicados dependentes [54], fazendo com que a estratégia menor n.p. selecione uma grande proporção de caminhos não executáveis. Estes dados da função *pstr* devem ser considerados, pois este tipo de não executabilidade não possui características fortes que facilitem a sua determinação estaticamente. Entretanto, o pouco número de caminhos gerados para a rotina *pstr* não influencia a conclusão sobre qual estratégia seleciona o maior número de caminhos executáveis gerados para o programa cal.

Como a rotina *main* do programa *cal*, possui um laço com o número de entradas determinado estaticamente, foram realizados dois tipos de medição, conforme a Etapa 4 da metodologia descrita na Seção 5.1.1. Os dados referentes à medição considerando o critério (b.i) da Seção 5.1.1 são apresentados no Apêndice C.

Na Tabela 5.2, referente ao programa *comm*, 5 entre 6 módulos têm como resultado o empate entre as três estratégias. Este resultado é devido a estes cinco módulos possuírem estruturas de controle simples (ver Apêndice B), ocasionando a geração de poucos caminhos. Detalhadamente tem-se que: para os módulos *openfil* e *wr* é gerado apenas um caminho para cada elemento requerido, tanto para o critério todos-potenciais-usos quanto para todos-ramos (Tabela 5.6); para os módulos *compare*, *copy* e *rd* tem-se vários elementos requeridos onde não havia cinco caminhos para serem gerados. Isto se deve à limitação do número de vezes que é permitida a entrada num ciclo.

Ainda na Tabela 5.2, a estratégia maior n.p. é melhor para a rotina *main*, influenciando diretamente o resultado total porque esta função possui muitos elementos requeridos (250 para todos-potenciais-usos e 27 para todos-ramos), ocasionando a geração de muitos caminhos (total de 1318 caminhos para cada estratégia, para todos-potenciais-usos).

Quanto à rotina *main* do programa *comm*, algumas considerações devem ser feitas sobre o maior número de caminhos executáveis ter sido selecionado pela estratégia maior n.p. (57,61% para o critério todos-potenciais-usos, como é possível visualizar na Tabela 5.11). Para estas con-

siderações utilizou-se o conhecimento do código da rotina *main* que teve algumas características destacadas na Seção 5.1.1. Nesta rotina, existe um laço do tipo "while", cuja condição é infinita, ou seja, sempre verdadeira (condição "1" na linguagem C). A única maneira de sair deste laço é executar uma função de saída do programa ("exit(0)"), que está aninhada sob vários predicados, que por sua vez estão aninhados sob este laço infinito. Desta maneira, os únicos caminhos executáveis são aqueles que saem do laço sem ser pela condição do "while(1)". A estratégia menor n.p. seleciona caminhos que nunca entram no laço, ou seja, caminhos não executáveis, enquanto a estratégia maior n.p. seleciona caminhos maiores que entram no laço e saem deste através da função exit(0), ou seja, executáveis em sua grande maioria.

Nas Tabelas 5.3 e 5.4, percebe-se na última coluna ("Maior Número de Executáveis") a predominância do empate entre as estratégias. A estratégia menor n.p. seleciona maior número de caminhos executáveis em 1 dos 4 módulos do program *entab* e 1 dos 2 módulos do programa *expand*. Este grande número de módulos com resultados empatados deve-se à simplicidade dos programas (ver Apêndice B), como na análise realizada anteriormente para os módulos do programa *comm*. Como um exemplo, tem-se as rotinas *main* dos programas *entab* e *expand* que possuem apenas um bloco de comando, com apenas uma chamada de função.

A Tabela 5.5 apresenta o empate entre as três estratégias para a rotina *jan1* e o valor de "Maior" para o módulo *pstr*. Para estas rotinas vale a análise apresentada para a Tabela 5.1 referente ao critério todos-potenciais-usos. Para o programa *cal*, 1 entre 4 módulos apresentou a estratégia menor n.p. como aquela que seleciona maior número de caminhos executáveis. Outra rotina, *main*, apresentou um empate literal entre as estratégias menor n.p. e caminhos aleatórios, no entretanto deve-se observar um empate técnico entre as três estratégias (1,88% de diferença).

Ainda em relação ao programa *cal*, percebe-se que, comparando as linhas "cal" das Tabelas 5.11 e 5.12, a estratégia menor n.p. ("Menor") teve uma diminuição na seleção de caminhos executáveis para o critério todos-ramos (Tabela 5.12), devido ao aumento do número de caminhos executáveis para a estratégia maior n.p., para a rotina *main* para este critério. Isto aconteceu porque os caminhos exigidos para cobrir arcos foram menores e coincidiram com os

selecionados pela estratégia menor n.p. para o critério todos-ramos.

A análise dos resultados do programa *comm*, da Tabela 5.6 para o critério todos-ramos não é diferente da realizada para o critério todos-potenciais-usos. Comparando as linhas "comm" das Tabelas 5.11 e 5.12 percebe-se uma diminuição da diferença entre as estratégias maior n.p. e menor n.p. em relação ao critério todos-potenciais-usos para todos-ramos, de 37,73% para 19,84%. Isto decorre do fato do critério todos-ramos exigir menos elementos requeridos para a função *main*, gerando-se menor número de caminhos em comparação ao critério todos-potencias-usos.

A função *settabs*, apresentada na Tabela 5.7, é simples (ver Apêndice B) e possui, dos caminhos gerados,, apenas um caminho não executável. Devido ao baixo número e reduzida complexidade dos elementos requeridos pelo critério todos-ramos, este caminho não foi selecionado pela estratégia maior n.p.; no entanto, foi selecionado por esta estratégia para o critério todos-potenciais-usos, como apresentado na Tabela 5.3. Isto provocou, na Tabela 5.3, o resultado da última coluna como "Maior". Considera-se este resultado um empate técnico entre as três estratégias para esta rotina, pois a diferença entre o número de caminhos executáveis selecionados pelas estratégias menor n.p. e aleatória e o número de caminhos executáveis selecionados pela estratégia maior n.p. foi de apenas um caminho, o que representa 6,67% do total de caminhos gerados.

Os resultados do experimento utilizando o programa *expand*, referente à Tabela 5.8 foram próximos aos da Tabela 5.4.

As Tabelas 5.9 e 5.10, apresentam a totalização geral dos programas e mostram a estratégia que selecionou maior número de caminhos executáveis total: a estratégia maior n.p.. Isto acontece pela grande quantidade de caminhos gerados para o programa *comm* (mais especificamente a rotina *main*). É percebido que todos os outros programas apresentaram caminhos que validaram a estratégia menor n.p.. Como o caso especial (laço infinito com saída do laço sob vários predicados) encontrado na rotina *main* do programa *comm*, é passível de detecção e tratamento estaticamente, geraram-se outras duas tabelas, e conseqüentes gráficos, de totalização geral, sem a contagem de caminhos referente à esta rotina. As Tabelas 5.13 e 5.14, e respectivamente

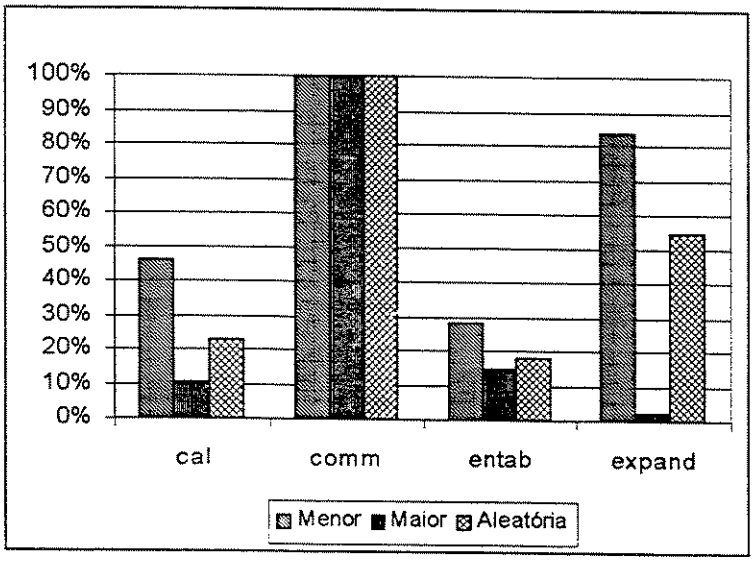

Figura 5.3: Porcentagem de caminhos executáveis - critério todos-potenciais-usos.

os gráficos mostrados nas Figuras 5.3 e 5.4, permitem a comparação e visualização dos valores alcançados no programa *comm*, sem a influência deste caso especial.

Nas Tabelas 5.13 e 5.14, o programa *comm* apresentou empate entre as três estratégias avaliadas. Isto gera um resultado geral favorável à estratégia menor n.p.. Estes resultados vêm mostrar que, tratados os casos especiais detectáveis estaticamente, a aplicação da estratégia de seleção de caminhos menor n.p. mostrou-se melhor na seleção de caminhos requeridos pelos critérios todos-potenciais-usos e todos-ramos pois selecionam em termos de porcentagem um maior número de caminhos executáveis. Para os resultados desse experimento, também se confirma que quanto menor o número de predicados em um caminho maior a probabilidade de ele ser executável.

5.2 Eficácia

A segunda aplicação do módulo Poke-Paths teve como objetivo propôr uma metodologia de experimento para observar o uso da estratégia menor número de predicados e a eficácia do processo de teste. Isto porque Yates e Malevris [62, 32] propuseram a estratégia menor número de predicados visando a diminuição do número de caminhos não executáveis selecionados, mas

Tabela 5.9: Total de caminhos - critério todos-potenciais-usos.

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
cal	319	374	70	623	157	536	*Menor*
comm	453	865	1313	5	513	805	*Maior*
entab	86	220	45	261	55	251	*Menor*
expand	109	21	3	127	71	59	*Menor*
total	967	1480	1431	1016	796	1651	*Maior*
total %	39,52	60,48	58,48	41,52	32,53	67,47	*Maior*

Tabela 5.10: Total de caminhos - critério todos-ramos.

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
cal	74	67	43	98	52	89	*Menor*
comm	99	79	173	5	101	77	*Maior*
entab	22	35	17	40	20	37	*Menor*
expand	30	5	2	33	19	16	*Menor*
total	225	186	235	176	192	219	*Maior*
total %	54,74	45,26	57,18	42,82	46,72	53,28	*Maior*

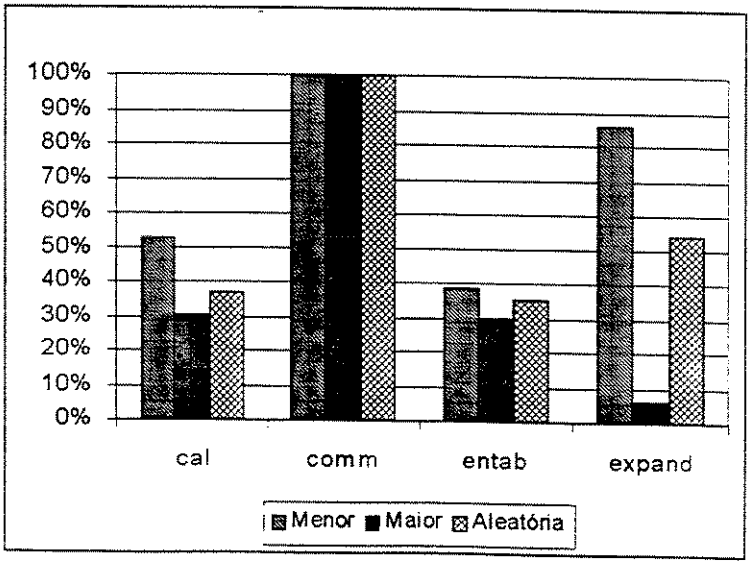

Figura 5.4: Porcentagem de caminhos executáveis - critério todos-ramos.

Tabela 5.11: Valores percentuais de caminhos executáveis, por estratégia, para o critério todos-potenciais-usos

	Menor %	*Maior %*	*Aleatória %*	Total
cal	58,42	12,82	28,75	546
comm	19,88	57,61	22,51	2279
comm*	33,33	33,33	33,33	207
entab	46,24	24,19	29,57	186
expand	59,56	1,64	38,80	183
total	30,28	44,80	24,92	3194
total*	51,96	16,67	31,37	1122

Tabela 5.12: Valores percentuais de caminhos executáveis, por estratégia, para o critério todos-ramos

	Menor %	*Maior %*	*Aleatória %*	Total
cal	43,79	25,44	30,77	169
comm	26,54	46,38	27,08	373
comm*	33,33	33,33	33,33	129
entab	37,29	28,81	33,90	59
expand	58,82	3,92	37,25	51
total	34,51	36,04	29,45	652
total*	41,42	25,74	32,84	408

sem avaliar sua eficácia. A questão que se pode colocar é se existe a redução da eficácia selecionando-se, entre dois caminhos, aquele com menor número de predicados. Parece intuitivo que quanto maior o número de predicados envolvidos aumenta a probabilidade de se revelar defeitos no programa.

A metodologia apresentada foi aplicada em um programa já utilizado no experimento anterior, o programa *cal*. Para este programa haviam sido derivadas vinte versões diferentes por Wong [61] e utilizadas em vários experimentos de modo a avaliar a eficácia [61, 55] de *critérios de teste*.

Nesta seção, serão apresentados a metodologia, sugerida e aplicada, e os resultados observados com esta aplicação.

Tabela 5.13: Total de caminhos - critério todos-potenciais-usos (excluída a função *main* do programa *comm*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
cal	319	374	70	623	157	536	*Menor*
comm	69	0	69	0	69	0	*Menor Maior Aleatória*
entab	86	220	45	261	55	251	*Menor*
expand	109	21	3	127	71	59	*Menor*
total	583	615	187	1011	352	846	*Menor*
total %	48,66	51,34	15,61	84,39	29,38	70,62	*Menor*

Tabela 5.14: Total de caminhos - critério todos-ramos (excluída a função *main* do programa *comm*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
cal	74	67	43	98	52	89	*Menor*
comm	43	0	43	0	43	0	*Menor Maior Aleatória*
entab	22	35	17	40	20	37	*Menor*
expand	30	5	2	33	19	16	*Menor*
total	169	107	105	171	134	142	*Menor*
total %	61,23	38,77	38,04	61,96	48,55	51,45	*Menor*

5.2.1 Metodologia

O objetivo desta metodologia é permitir a verificação da eficácia de estratégias de seleção de caminhos quando aplicadas à geração automática de caminhos, aplicando o módulo Poke-Paths e a ferramenta Poke-Tool. Esta metodologia foi baseada nos experimentos realizados por Wong *et al.*[60, 61] e Vergilio [55].

Esta metodologia requer como entrada um programa e um determinado número de versões incorretas. As versões incorretas devem possuir um único erro introduzido. Sua meta é verificar a capacidade de um conjunto de casos de teste em revelar os defeitos introduzidos.

A metodologia propõe as seguintes etapas:

1. Geração automática de caminhos completos. Nesta geração de caminhos, as estratégias de seleção de caminhos são aplicadas. Observa-se que o conjunto de caminhos selecionados

não satisfaz necessariamente os critérios utilizados, visto que o objetivo deste experimento não é avaliar a eficácia dos critérios.

2. Geração de dados de teste para cada caminho completo executável gerado no passo anterior. Para a determinação dos dados de teste que executem caminhos que passem por laços determinados estaticamente, adota-se o critério (b.ii) apresentado na Seção 5.1.1. Nessa etapa, também são determinados manualmente os caminhos não executáveis.

3. Execução dos dados de teste, encontrados na etapa anterior, para as versões incorretas do programa em questão, sendo registrados seus resultados.

4. Comparação dos resultados obtidos com os resultados do programa original (versão correta).

5.2.2 Aplicação da Metodologia

Utilizou-se o programa *cal* para a aplicação desta metodologia. Em [61] foram derivadas vinte versões incorretas deste programa. As diferenças entre as versões incorretas e correta são apresentadas no Apêndice D. Das vinte versões com defeito inserido, 6 versões possuem erro de domínio e 14 versões erro de computação [55].

O programa *cal* foi utilizado no experimento descrito na Seção 5.1, para o qual um conjunto de caminhos completos foi gerado. Para este experimento, os mesmos caminhos completos foram utilizados.

Na próxima seção são apresentados alguns resultados interessantes observados com esta aplicação e que podem basear análises a serem realizadas em futuros experimentos.

Resultados

Esta observação empírica obteve resultados intermediários, que são apresentados no Apêndice D e os resultados finais que são apresentados na Tabela 5.15. Esta tabela apresenta, respectivamente, dois valores em suas células: os números de casos de teste encontrados para os caminhos selecionados pelas estratégias menor n.p., "Menor", maior n.p., "Maior" e "Aleatória", para os

elementos requeridos pelos critérios todos-potenciais-usos e todos-ramos; e o número de defeitos revelados por estes casos de teste.

Tabela 5.15: Número de casos de teste por erros revelados

	Menor		*Maior*		*Aleatória*	
	no.dados	no.def.	no.dados	no.def.	no.dados	no.def.
todos-ramos	9	20	8	20	8	20
todos-potenciais-usos	10	19	3	15	14	19

Por estes resultados percebe-se uma diferença significativa entre o número de defeitos revelados pelos dados de teste selecionados com o auxílio da estratégia maior n.p. e as outras estratégias. Esta diferença é negativa para esta estratégia, que conseguiu selecionar caminhos que revelassem apenas 15 defeitos contra as outras estratégias que revelaram 19, para o critério todos-potenciais-usos. Isto provavelmente ocorreu pelo maior número de caminhos não executáveis gerados pela estratégia maior número de predicados, o que fez com que se gerasse menor número de dados de teste, conseqüentemente, diminuindo as chances de revelar defeitos.

Percebe-se que os números de casos de teste encontrados e defeitos revelados para o critério todos-potenciais-usos são geralmente menores do que para o critério todos-ramos. Isto aconteceu porque não se objetivou garantir a cobertura de 100% dos elementos requeridos pelos critérios todos-potenciais-usos e todos-ramos. Detectou-se um maior número deste caso para elementos requeridos pelo critério todos-potenciais-usos, provavelmente por requererem caminhos maiores (conseqüentemente com maior número de predicados). Para a cobertura destes elementos deveriam ter sido gerados maior número de caminhos para cada elemento requerido, para que se alcançasse a seleção de caminhos executáveis.

Percebe-se, com o auxílio das tabelas apresentadas no Apêndice D, que para as estratégias menor n.p. e aleatória o erro não revelado foi de computação, enquanto para a estratégia maior n.p. quatro foram de predicado e um (o mesmo que das outras estratégias) foi de computação.

Este experimento forneceu um indicativo positivo sobre a capacidade de revelar defeitos da estratégia menor n.p., pois, para esse programa, não foi percebida diminuição da eficácia utilizando a estratégia em questão, nem mesmo comparando-se com a estratégia aleatória.

Entretanto, não é possível concluir que o número de predicados em caminhos influenciam na eficácia em revelar defeitos ou quais influências exercem. Outros experimentos devem ser conduzidos de modo a alcançar alguma conclusão a respeito e para quais tipos de programas e tipos de erros que influenciariam.

5.3 Considerações Finais

Este capítulo apresentou resultados de duas aplicações utilizando o módulo Poke-Paths, observando o uso da estratégia menor número de predicados: um referente à executabilidade dos caminhos selecionados por esta estratégia e outro referente à capacidade de revelar defeitos de dados de teste gerados para executar estes caminhos.

Percebeu-se através do experimento "quanto ao número de caminhos não executáveis", que:

- Desconsiderando-se um caso especial apresentado (rotina *main* do programa *comm*), a estratégia menor número de predicados seleciona caminhos com maior probabilidade de serem executáveis, para os programas avaliados. É importante ressaltar que a escolha de estratégias de seleção de caminhos deve depender das características dos programas testados.

- Observou-se que, em casos de programas que possuam laço infinito com o comando de saída do laço aninhado sob vários predicados, a estratégia menor número de predicados não possui maior probabilidade de selecionar caminhos executáveis, de acordo com análise apresentada na Seção 5.1.3. São necessários análise estática e tratamentos a respeito de laços infinitos e laços com número de entrada determinado estaticamente. Estas determinações e tratamentos são passíveis de automatização.

- De acordo com as aplicações, percebeu-se que a simplicidade do código e o limite imposto para a entrada em ciclos, que é uma necessidade do algoritmo, aliado às restrições dos elementos requeridos, provocam poucas opções de caminhos no conjunto inicial de caminhos a serem selecionados. Esta observação sugere que, para a geração de caminhos de rotinas extremamente simples, é indiferente qual estratégia de seleção de caminhos adotar.

- Após a caracterização de padrões de não executabilidade de caminhos, executou-se o módulo Pokepadrão, um programa de comparação de padrões de não executabilidade com caminhos gerados pelo módulo Poke-Paths. Esses módulos fazem parte da ferramenta Poke-Tool e foram descritas em capítulos anteriores. O grande número de caminhos que não foram detectados como não executáveis tiveram que ser analisados manualmente para a detecção de sua executabilidade. Estas duas atividades, a caracterização de padrões e a determinação manual da executabilidade de caminhos, mostraram-se como as mais desgastantes e demoradas.

É interessante observar que a segunda aplicação, quanto à eficácia, forneceu um indicativo a respeito da capacidade dos caminhos selecionados por estratégias em revelar defeitos. Percebe-se, no estudo realizado, que a eficácia dos dados de teste referentes à estratégia menor número de predicados não é menor que a das outras estratégias. Os resultados aqui apresentados constituem-se numa motivação para que estudos mais detalhados com relação à influência dessa estratégia na eficácia sejam realizados.

Capítulo 6

Conclusões

Este trabalho identificou as linhas gerais de pesquisa de estratégias de seleção de caminhos e geração de caminhos completos para satisfazer critérios estruturais de teste. Levantou-se na literatura trabalhos sobre a geração automática de caminhos, como os dois trabalhos de Bertolino e Marré, [3, 4], que sugerem a utilização de algoritmos que implementam as estratégias menor número de predicados e minimização de casos de teste. Estes algoritmos geram caminhos para critérios todos-ramos e todos-usos.

Neste trabalho, foram propostas novas estratégias de seleção de caminhos, baseadas em características do software. As estratégias propostas são utilizadas para selecionar os "melhores" caminhos a partir de um conjunto inicial de caminhos que cobrem um dado elemento requerido por algum critério de teste (adequação e/ou seleção). Como nenhuma análise específica de cobertura de teste do programa foi realizada para as estratégias de seleção de caminhos, não é recomendada sua aplicação sem a utilização de critérios, não sendo possível garantir a qualidade do teste de software.

As estratégias de seleção de caminhos provocam reflexos na atividade de geração de dados de teste, buscando-se facilitar esta atividade e diminuir seus custos. Alguns destes reflexos:

- propiciar a aplicação de algoritmos mais simplificados para a execução simbólica ou facilitar a geração de dados manualmente, selecionando caminhos mais simples, ou seja, de menor complexidade;

- tornar mais eficaz a aplicação dos critérios estruturais, aumentando a probabilidade em

detectar defeitos através da seleção de caminhos mais complexos e com maior testabilidade e em selecionar caminhos com maior probabilidade de serem executáveis [62, 32];

Nesta dissertação, foi proposta uma estrutura de representação de estratégias de seleção de caminhos baseada em teoria dos grafos, onde os custos associados aos nós podem representar a complexidade, testabilidade, não executabilidade, ou outras características desejadas, como o tempo, esforço ou custo do teste de software. Desta forma, sugere-se que o problema fique reduzido a selecionar o maior ou menor caminho que traduza a necessidade do testador. Esta estrutura apresenta como vantagens: permitir "dividir para conquistar" as características associadas ao código, de modo a serem selecionados caminhos que as possuam em maior ou menor grau; permitir rápida e fácil automatização, aplicando-se uma teoria já bem definida de grafos. Como destacado na Tabela 3.1, observa-se que a utilização desta estrutura necessita da aplicação de características que possam ser contadas diretamente a partir do código. Isto implica que características subjetivas podem ser utilizadas apenas se houver um mapeamento para valores numéricos que possam ser associados a blocos de comandos.

As estratégias de seleção de caminhos propostas, mapeadas nessa estrutura de representação de estratégias, não têm como objetivo minimizar o número de caminhos gerados para todo o teste estrutural. O objetivo destas estratégias é tornar este teste mais efetivo provendo mecanismos de redução de caminhos não executáveis, permitindo a seleção de caminhos aparentemente mais eficazes ou mais simples para a geração de dados, contribuindo para a redução de custo e tempo total do teste de software.

Apesar das estratégias terem sido ilustradas utilizando os critérios todos-potenciais-usos e todos-ramos, estas podem ser utilizadas com qualquer critério baseado em caminhos.

Tem-se como uma das contribuições deste trabalho a implementação de um módulo de extensão à ferramenta de teste Poke-Tool, que permite a geração automática de caminhos completos, com economia expressiva de esforço e tempo. Os caminhos gerados cobrem elementos requeridos pelos critérios todos-nós, todos-arcos, todos-potenciais-du-caminhos, todos-potenciais-usos e todos-potenciais-usos/du. Este módulo, o Poke-Paths, pode ser estendido a outros critérios baseados em fluxo de controle e em fluxo de dados, com alterações na leitura

dos arquivos de entrada.

Este módulo implementa dois tipos de seleção de caminhos: uma determinada por uma estratégia de seleção de caminhos que possui qualquer definição no formato $S = (p(n), e(\pi))$, descrito neste texto, e outra aleatória, ou seja, que não se utiliza de nenhuma estratégia de seleção de caminhos. A implementação da estrutura de representação de estratégias permite a incorporação de diferentes características do processo de desenvolvimento e do produto de software nas atividades de teste de software.

Segundo Bertolino e Marré [3], ferramentas de geração automática de caminhos devem incorporar meios de minimizar a seleção de caminhos não executáveis. Com base nestes objetivos, no módulo Poke-Paths foi implementada a estratégia de seleção de caminhos menor número de predicados. Este módulo propicia a geração de caminhos incorporada às heurísticas de detecção de elementos requeridos não executáveis, implementadas pelo ambiente de teste Poke-Tool. No Capítulo 4, foi descrito um procedimento completo de teste que visa a diminuição de caminhos não executáveis, utilizando a ferramenta Poke-Tool e o módulo Poke-Paths.

A implementação do módulo Poke-Paths foi validada através de dois procedimentos de aplicação, onde foi possível observar alguns resultados interessantes quanto à capacidade da estratégia menor número de predicados em selecionar maior número de caminhos executáveis e em revelar defeitos.

Na primeira aplicação, foram detectadas duas características de programas que fazem com que a estratégia menor número de predicados não possua maior probabilidade de selecionar caminhos executáveis: laço infinito com o comando de saída do laço aninhado sob vários predicados e laços com alto número de execuções determinado estaticamente. A primeira já havia sido considerada por autores [62, 32, 3] anteriormente. A segunda foi levantada através desta aplicação. Sugere-se, para ser possível a aplicação desta estratégia com maior êxito, detectar e tratar estaticamente estes tipos de estrutura e outras que vierem a ser descobertas. Ainda, é importante ressaltar que a escolha das estratégias de seleção de caminhos depende das características dos programas testados.

A segunda avaliação empírica mostrou-se como um protótipo de aplicação de uma metodo-

logia de avaliação da eficácia de estratégias de seleção de caminhos. Os resultados alcançados constituem-se uma motivação para que estudos mais detalhados, considerando a eficácia da estratégia menor número de predicados, sejam realizados e que estes estudos a validem como uma estratégia que permite reduzir efeitos causados pelos caminhos não executáveis nas atividades de teste, sem redução da eficácia.

6.1 Trabalhos Futuros

Alguns experimentos podem ser realizados para validar a estrutura de representação de estratégias, suas instanciações de acordo com características diversas, e a estrutura do módulo Poke-Paths:

- Experimento a ser conduzido com a estratégia menor número de predicados e de acordo com a metodologia de minimização de caminhos não executáveis, descrita no Capítulo 4. Considerando um conjunto maior de programas o experimento poderá avaliar a quantidade de caminhos não executáveis gerados e levantar as estruturas dos programas que dificultam a aplicação dessa estratégia;

- Experimento a ser conduzido para avaliar a eficácia dos caminhos gerados, utilizando um *benchmark* maior com introdução de defeitos;

- Experimento para avaliar e comparar, em termos de número de caminhos executáveis e eficácia, as diferentes estratégias propostas neste trabalho, com adaptação ao módulo de geração de dados de teste desenvolvido por Bueno [7], que gera dados de teste a partir de caminhos completos fornecidos como entrada;

- Comparar resultados dos experimentos propostos para os critérios todos-potenciais-usos e todos-ramos, realizando um trabalho específico de comparação dos dois critérios a respeito da eficácia de estratégias de seleção de caminhos associadas a critérios;

Alguns trabalhos quanto à implementação do módulo Poke-Paths podem ser realizados, provendo maior capacidade a este módulo:

- Implementação de algoritmos que encontrem caminhos ou rotas alternativas, de modo a acrescentar a capacidade de minimização de casos de teste;
- Estudo sobre a incorporação de padrões de não executabilidade ao algoritmo de geração de caminhos, com o objetivo de descartar caminhos não executáveis em tempo de execução;
- Incorporação de outros algoritmos para encontrar menores e maiores caminhos;
- Incluir a capacidade de flexibilizar o número de entradas em laços, de acordo com o tipo de estruturas contidas no programa em teste, o que resultaria num tratamento para laços com número de entradas determinado estaticamente;
- Estudo e implementação da análise estática e de tratamentos para laços infinitos, de modo a facilitar a determinação de executabilidade de caminhos que contenham estas estruturas.

Bibliografia

[1] H. et al. Agrawal. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, July 1998.

[2] A.J. Albrecht. Measuring application development productivity. *Proc. IBM Application Development Symposium*, 2:83–92, October 1979.

[3] A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer program. *IEEE Trans. on Software Engineering*, 20(12):885–899, December 1994.

[4] A. Bertolino and M. Marré. Unconstrained duas and their use in achieving all-uses coverage. In *ISSTA '96- ACM*, San Diego,CA, 1996.

[5] A. Bertolino and L. Strigini. On the use of testability measures for dependability assessment. *IEEE Transactions on Software Engineering*, 22(2):97–108, February 1996.

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

[7] P.M.S. Bueno. Geração automática de dados e tratamento de não executabilidade no teste estrutural de software. Master's thesis, DCA/FEEC/Unicamp, Campinas/SP, Junho 1999.

[8] P.M.S. Bueno, M.L. Chaim, J.C. Maldonado, M. Jino, and P.R.S. Vilela. Manual do usuário da Poke-Tool. Technical report, DCA/FEEC/Unicamp, Campinas, SP, 1995.

[9] P.M.S. 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, janeiro 1997.

[10] Software Engineering Institute Carnegie Mellon University. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, 1th edition, 1995.

[11] M. L. Chaim. Poke-tool - uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. Master's thesis, DCA/FEE/Unicamp, Campinas/SP, Abril 1991.

[12] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models.* Benjamin/Cummings Publishing Company, Inc., 1986.

[13] H.E. Dunsmore and J.D. Gannon. Analysis of the effects of programming factors on programming effort. In V.R.Basili, editor, *Tutorial on Models and Metrics for Software Management and Engineering*, pages 93–105. IEEE Catalog No. EHO-167-7, Computer Society Press, New York, 1980.

[14] Chong E.I., Maddila S., and Morley S. On finding single-source single-destination k shortest paths. In Journal of Computing and Information(JCI), editors, *Proceedings of Seventh International Conference of Computing and Information*, pages 40–47, Trent University, Peterborough, Ontario, Canada, July 1995.

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

[16] P. G. Frankl. *The Use of Data Flow Information for Selection and Evaluation of Software Test Data.* PhD thesis, New York University, October 1987.

[17] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[18] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, New York, NY, 1979.

[19] M. Halstead. *Elements of Software Science.* North-Holland, 1977.

[20] W. Harrison and et alii. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3), March 1981.

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

[22] P.M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3), November 1976.

[23] J.R. Horgan and S. London. Data flow coverage and the c language. *Proceedings Fourth Symposium Software Testing, Analysis, and Verification*, October 1991.

[24] W.E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Software Engineering*, 24(5), May 1975.

[25] C. Jones. *Applied Software Measurement.* McGraw-Hill, 1991.

[26] J.W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3), May 1983.

[27] J.R. Levine, Mason T., and Brown D. *lex & yacc.* O'Reilly & Associates, Inc., May 1990.

[28] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* PhD thesis, DCA/FEE/Unicamp, 1991.

[29] J.C. Maldonado, M.L. Chaim, and Jino M. Seleção de casos de testes baseada nos critérios potenciais usos. In *II Simpósio Brasileiro de Engenharia de Software*, pages 24–35, Canela, RS, Brasil, Outubro 1988.

[30] J.C. Maldonado, M.L. Chaim, and Jino M. Feasible potential use criteria analysis. Technical Report RT/DCA-001/89, DCA/FEE/Unicamp, Campinas, SP, Brasil, 1989.

[31] J.C. et al. Maldonado. *Aspectos Teóricos e Empíricos de Teste de Cobertura de Software.* Notas Didáticas, ICMSC/USP, São Carlos, SP, Brazil, June 1998.

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

[33] E.Q.V Martins, M.M.B Pascoal, and J.L.E. Santos. The k shortest loopless paths problem. *submetido*, June 1998.

[34] E.Q.V Martins, M.M.B Pascoal, and J.L.E. Santos. The k shortest paths problem. *submetido*, June 1998.

[35] T.A. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[36] E.F. Miller Jr. and R.A. Melton. Automated generation of testcase datasets. *ACM Sigplan Notices*, Vol. 10(6):51–58, June 1975.

[37] J.C. Munson and Khoshgoftaar. *Software Metrics for Reliability Assessment*, volume In Handbook of Software Reliability Engineering. McGraw-Hill, New York, 1996. chapter 12, pages 493-529.

[38] G.J. Myers. *The Art of Software Testing.* Wiley, 1979.

[39] Glenford J. Myers. An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices*, 12(10):61–64, October 1977.

[40] B. A. Nejmeh. Npath: A measure of execution path complexity and its applications. *Comm. of th ACM*, 31(2):188–200, February 1988.

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

[42] P. Oman and J. Hagemeister. Constructing and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266, March 1994.

[43] L.M. Peres, S.R. Vergilio, M. Jino, and J.C. Maldonado. Aspectos de seleção de caminhos para cobertura de critérios estruturais de teste. *Workshop do Projeto Validação e Teste de Sistemas de Operação*, January 1997.

[44] L.M. Peres, S.R. Vergilio, M. Jino, J.C. Maldonado, A.M.A. Price, and J.B. Silva. Critérios potenciais usos: Um mecanismo para a definição de métricas de complexidade de software. *Workshop do Projeto Validação e Teste de Sistemas de Operação*, January 1997.

[45] R.B. Pressman. *Software Engineering: A Practitioner's Approach.* Addison-Wesley, 4th edition, 1995.

[46] A.M.A. Price and A. Zorzo. Visualizando o fluxo de controle de programas. *IV Simpósio Brasileiro de Engenharia de Software, Águas de São Pedro, SP*, Outubro 1990.

[47] S. Rapps and E.J. Weyuker. Data flow analysis techniques for test data selection. In *Proc. Int. Conf. Software Engineering*, Tokio, Japão, September 1982.

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

[49] V.Y. Shen, S.D. Conte, and H.E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. *IEEE Transactions on Software Engineering*, 9(2):155–165, March 1983.

[50] M.L. Shooman. *Software Engineering.* New York, 1983.

[51] J.B. Silva and A.M.A. Price. Métrica de complexidade de software baseada em critério de seleção de caminhos de teste. *VIII Simpósio Brasileiro de Engenharia de Software, Curitiba*, pages 471–485, 1994.

[52] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28:157–163, 1988.

[53] 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 Conferência Latinoamericana de Informática*, Buenos Aires, Argentina, Agosto 1993.

[54] S.R. Vergilio. Caminhos não executáveis: Caracterização, previsão e determinação para suporte ao teste de programas. Master's thesis, DCA/FEEC/Unicamp, Campinas, SP, janeiro 1992.

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

[56] S.R. Vergilio, J.C. Maldonado, and M. Jino. Caminhos não executáveis na automação das atividades de teste. *VI Simpósio Brasileiro de Engenharia de Software*, pages 343–356, novembro 1992.

[57] J. Voas, L. Morell, and K. Liller. Predicting where faults can hide from testing. *IEEE - Software*, pages 41–47, March 1991.

[58] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1371, September 1988.

[59] E.J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19(12), 1984.

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

[61] 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.

[62] D. F. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. *ACM SIGSOFT Software Engineering Notes*, 14(8):48–54, December 1989.

Apêndice A

Detalhes de Utilização do Módulo Poke-Paths

Este apêndice tem como objetivo apresentar alguns detalhes de utilização do programa Poke-Paths, um módulo de extensão à ferramenta de teste Poke-Tool. Este módulo foi descrito no Capítulo 4.

A ferramenta Poke-Tool escreve seus arquivos dentro de um diretório criado pelo usuário, que tem como nome o programa em instrumentação. O módulo Poke-Paths deve ser executado a partir deste diretório, que contém todos os arquivos e diretórios referentes ao *programa P*, que terá seus caminhos gerados. Isto quer dizer que este diretório deverá conter os arquivos de entrada do Poke-Paths, sendo que os arquivos referentes ao *programa P* devem estar localizados diretamente em sua raiz e os referentes à *função f* em diretório com o nome da função a ter seus caminhos gerados.

A Seção A.1 apresenta a descrição detalhada das entradas de execução do módulo Poke-Paths. Um exemplo de arquivo gerado por este módulo Poke-Paths é apresentado na Seção A.2. A Seção A.3 apresenta um arquivo gerado após a determinação de executabilidade utilizando o módulo Poke-Padrão, da ferramenta Poke-Tool. Os arquivos de saída apresentados foram gerados pelo experimento descrito na Seção 5.1, do Capítulo 5.

A.1 Descrição da Entrada

A seguir são apresentados o formato dos dados de entrada por parte do usuário e explicações a respeito de cada opção.

pokepaths -d<função> -[nos | arcs | pu | pdu] -[all | l<lista> | i'<x> to <y> '|u'<elemento> '] -e<estratégia> [-m] [-c<número>] [-r<número>] -o<arquivo de saída>

onde:

-d<função>: <função> é o nome da função em teste a ter seus caminhos gerados;

Esta opção permite a localização dos arquivos de entrada da Poke-Tool, referentes à <função>, que se deseja gerar os caminhos completos.

Há um subdiretório <função> num nível abaixo do diretório <programa>, onde estão os arquivos *grafodef.tes* e *<funcao>.gfc*. Estes arquivos são utilizados para a criação do grafo def, a principal estrutura de dados deste módulo.

-[nos|arcs|pu|pudu|pdu]:

Através do critério de teste de software determinado por esta opção, é escolhido o arquivo de elementos requeridos que será utilizado para a geração de caminhos completos. Ou ainda, caso seja fornecido o elemento requerido pela opção "-u'<elemento>"', esta opção define as regras de divisão em subcaminhos (apresentadas no Capítulo 4). Esta opção permite as seguintes escolhas:

-nos: o arquivo é nos_grf.tes e o elemento requerido é do tipo nó;

-arcs: o arquivo é arcprim.tes e o elemento requerido é do tipo arco;

-pu: o arquivo é puassoc.tes e o elemento requerido é do tipo associação potencial uso;

-pdu: o arquivo é pdupaths.tes e o elemento requerido é do tipo potencial du-caminho;

-[all | l<lista>| i'<x> to <y>' | u"<elemento>"]

Esta opção permite que se requisite a este módulo a geração de caminhos completos para diversas quantidades de elementos: apenas um, todos elementos requeridos por um critério, todos de uma determinada lista e todos pertencentes a um determinado intervalo.

-all: gerar caminhos para todos os elementos requeridos pelo critério e contidos no arquivo de elementos requeridos determinado por um critério;

-l<lista>: gerar caminho somente para os elementos referenciados no arquivo texto <lista>. O conteúdo deste arquivo é uma lista de números ordenados, referente à ordem dos elementos no arquivo de elementos requeridos por um determinado critério. Como separação destes números inteiros que compõem a lista são permitidos: espaço em branco, vírgula, ponto e vírgula, tabulação e < *enter* > (nova linha);

-i'< x > to < y >': gerar caminhos somente para elementos que estão no intervalo x-y de números inteiros. Estes números representam a ordem dos elementos no arquivo de elementos requeridos de acordo com o critério entrado. Como exemplo tem-se a string: -i'5 to 10', sendo 5, 6, 7, 8, 9 e 10 os números correspondentes aos elementos requeridos desejados.

-u<elemento>: permite a geração de caminhos completos para um único elemento requerido. É fornecido como uma string. Ex.: -u'3', -u'(5,7)', -u'<2,(3,4),i>' , sendo 3 um nó, (5,7) um arco e <2,(3,4),i> uma associação. É importante ressaltar que o critério escolhido deve coincidir com o tipo de elementos fornecido como entrada.

-e<estratégia>: <estratégia> é um número inteiro indicando a estratégia de geração de

caminhos que é adotada, podendo assumir: "1" para a utilização da associação de pesos aos nós ($p(n)$ de S, segundo a definição do Capítulo 3), segundo a estratégia de seleção de caminhos número de predicados; "2", para a geração aleatória de caminhos; e "3" para a atribuição de pesos aos nós a ser realizada manualmente pelo usuário, podendo ser utilizada quaisquer estratégias de seleção de caminhos de teste apresentadas no Capítulo 3.

Para a opção -e1, a atribuição de pesos ao nó se dá realizando a leitura do arquivo < *programa* >.nli. Isto porque a Linguagem Intermediária, implementada na ferramenta Poke-Tool, possui a informação da quantidade de predicados contidos em um nó no grafo de fluxo de controle.

-m: indica qual $e(\pi)$ de S deve ser usado, segundo definições dadas no Capítulo 3, ou seja, se deve-se gerar caminhos com *menores* ou *maiores* pesos. Esta opção completa a implementação do par ordenado $S = (p(n), e(\pi))$ O *default* é encontrar os caminhos com *menores* pesos.

-c<número>: <número> é um inteiro indicando o número de caminhos completos que devem ser gerados, ou seja, corresponde ao k, de k-ésimos menor, maior ou aleatórios. Default = 1;

-r<número>: <número> é um inteiro indicando o número de vezes que é permitida a entrada no ciclo. Default = 1;

-o< *arquivo* >: opção que permite redefinir o nome e diretório do arquivo de saída. Um exemplo do arquivo de saída < *arquivo* > é apresentado no Apêndice C. Default = "cp<critério><estratégia>.tes";

A.2 Exemplo de Arquivo de Caminhos Completos Gerado por Poke-Paths

O arquivo a seguir é iniciado por um cabeçalho que contém as opções de entrada digitadas pelo usuário (apresentadas na Seção 4.3.1). Após este cabeçalho seguem-se vários conjuntos de caminhos completos gerados para cada elemento requerido, separados por uma linha preenchida por "#".

Para cada elemento requerido tem-se uma linha indicando seu número, o mesmo do arquivo de entrada, e um conjunto de caminhos completos. Após a identificação do número do elemento requerido, tem-se uma linha em branco e então tem-se uma linha referente ao caminho completo gerado e outra linha contendo dados referentes a este caminho. Os dados "peso", "tamanho" e "pred", referem-se respectivamente ao peso total do caminho (P_π, como no Capítulo 3), tamanho do caminho em número de nós e número total de predicados no caminho. As características "peso" e "pred" possuem o mesmo valor para este experimento. Após os caminhos completos, e seus dados, gerados para o elemento requerido, tem-se uma linha que apresenta o número de

caminhos completos gerados para o elemento em questão.

O exemplo a seguir é de um arquivo contendo os cinco (5) maiores caminhos completos, com restrição de duas (2) entradas em laço, que cubram elementos requeridos pelo critério todos-ramos para a função *main* do programa *cal*. Antes, porém, é apresentada a lista dos elementos requeridos que originaram este arquivo-exemplo.

```
ARCOS PRIMITIVOS DO MODULO main

  1) arco ( 1, 2)
  2) arco ( 3, 4)
  3) arco ( 5, 6)
  4) arco ( 5, 7)
  5) arco ( 7, 8)
  6) arco ( 7, 9)
  7) arco (11,12)
 ...


Caminhos Completos Gerados de acordo com a opcao:
pkteste -dmain -arcs -all -e1 -k -m -c5 -r2
        -omain/pkpaths/200199/geracao/arcsm1k52.tes

######################################################
numero do elemento requerido: 1

1 2 14 16 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 24 17 25
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 25
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 24 17 18 19 21 22 23 22 24 17 25
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 23 22 24 17 18 19 21 22 24 17 25
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 20 19 21 22 23 22 23 22 24 17 25
peso=11.00; tamanho=20; pred=11.00
numero de caminhos gerados para o elemento: 5

######################################################
numero do elemento requerido: 2

1 3 4 10 11 12 11 12 11 13 25
```

```
peso=5.00; tamanho=11; pred=5.00
1 3 4 10 11 12 11 13 25
peso=4.00; tamanho=9; pred=4.00
1 3 4 10 11 13 25
peso=3.00; tamanho=7; pred=3.00
numero de caminhos gerados para o elemento: 3

#####################################################
numero do elemento requerido: 3

1 3 5 6 25
peso=4.00; tamanho=5; pred=4.00
numero de caminhos gerados para o elemento: 1

######################################################
numero do elemento requerido: 4

1 3 5 7 9 10 11 12 11 12 11 13 25
peso=9.00; tamanho=13; pred=9.00
1 3 5 7 9 10 11 12 11 13 25
peso=8.00; tamanho=11; pred=8.00
1 3 5 7 9 10 11 13 25
peso=7.00; tamanho=9; pred=7.00
1 3 5 7 8 25
peso=6.00; tamanho=6; pred=6.00
numero de caminhos gerados para o elemento: 4

#####################################################
numero do elemento requerido: 5

1 3 5 7 8 25
peso=6.00; tamanho=6; pred=6.00
numero de caminhos gerados para o elemento: 1

#####################################################
numero do elemento requerido: 6

1 3 5 7 9 10 11 12 11 12 11 13 25
peso=9.00; tamanho=13; pred=9.00
1 3 5 7 9 10 11 12 11 13 25
peso=8.00; tamanho=11; pred=8.00
1 3 5 7 9 10 11 13 25
peso=7.00; tamanho=9; pred=7.00
numero de caminhos gerados para o elemento: 3
```

```
#####################################################
numero do elemento requerido: 7

1 3 5 7 9 10 11 12 11 12 11 12 11 12 11 12 11 13 25
peso=12.00; tamanho=19; pred=12.00
1 3 5 7 9 10 11 12 11 12 11 12 11 12 11 13 25
peso=11.00; tamanho=17; pred=11.00
1 3 5 7 9 10 11 12 11 12 11 12 11 13 25
peso=10.00; tamanho=15; pred=10.00
1 3 5 7 9 10 11 12 11 12 11 13 25
peso=9.00; tamanho=13; pred=9.00
1 3 4 10 11 12 11 12 11 12 11 12 11 12 11 13 25
peso=8.00; tamanho=17; pred=8.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: ...

...
```

A.3 Exemplo de Arquivo com Caminhos Completos com Executabilidade Determinada

Os arquivos obtidos após a etapa 4 da metodologia do experimento possuem o mesmo formato apresentado na seção anterior, com o acréscimo da expressão "–NEX" ao final da linha contendo o caminho completo, para caminhos completos não execu'taveis.

O exemplo a seguir refere-se ao mesmo exemplo apresentado na seção anterior, com os mesmos elementos requeridos, agora com os caminhos não executáveis identificados por "–NEX".

```
Caminhos Completos Gerados de acordo com a opcao:
pkteste -dmain -arcs -all -e1 -k -m -c5 -r2
        -omain/pkpaths/200199/geracao/arcsm1k52.tes

#####################################################
numero do elemento requerido: 1

1 2 14 16 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 25   --NEX
```

```
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 24 17 18 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 23 22 24 17 18 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 20 19 21 22 23 22 23 22 24 17 25
peso=11.00; tamanho=20; pred=11.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: 2

1 3 4 10 11 12 11 12 11 13 25
peso=5.00; tamanho=11; pred=5.00
1 3 4 10 11 12 11 13 25
peso=4.00; tamanho=9; pred=4.00
1 3 4 10 11 13 25   --NEX
peso=3.00; tamanho=7; pred=3.00
numero de caminhos gerados para o elemento: 3

#####################################################
numero do elemento requerido: 3

1 3 5 6 25
peso=4.00; tamanho=5; pred=4.00
numero de caminhos gerados para o elemento: 1

#####################################################
numero do elemento requerido: 4

1 3 5 7 9 10 11 12 11 12 11 13 25
peso=9.00; tamanho=13; pred=9.00
1 3 5 7 9 10 11 12 11 13 25
peso=8.00; tamanho=11; pred=8.00
1 3 5 7 9 10 11 13 25   --NEX
peso=7.00; tamanho=9; pred=7.00
1 3 5 7 8 25
peso=6.00; tamanho=6; pred=6.00
numero de caminhos gerados para o elemento: 4

#####################################################
numero do elemento requerido: 5

1 3 5 7 8 25
```

```
peso=6.00; tamanho=6; pred=6.00
numero de caminhos gerados para o elemento: 1

#####################################################
numero do elemento requerido: 6

1 3 5 7 9 10 11 12 11 12 11 13 25
peso=9.00; tamanho=13; pred=9.00
1 3 5 7 9 10 11 12 11 13 25
peso=8.00; tamanho=11; pred=8.00
1 3 5 7 9 10 11 13 25   --NEX
peso=7.00; tamanho=9; pred=7.00
numero de caminhos gerados para o elemento: 3

#######################################################
numero do elemento requerido: 7

1 3 5 7 9 10 11 12 11 12 11 12 11 12 11 12 11 13 25
peso=12.00; tamanho=19; pred=12.00
1 3 5 7 9 10 11 12 11 12 11 12 11 12 11 13 25
peso=11.00; tamanho=17; pred=11.00
1 3 5 7 9 10 11 12 11 12 11 12 11 13 25
peso=10.00; tamanho=15; pred=10.00
1 3 5 7 9 10 11 12 11 12 11 13 25
peso=9.00; tamanho=13; pred=9.00
1 3 4 10 11 12 11 12 11 12 11 12 11 12 11 13 25
peso=8.00; tamanho=17; pred=8.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: ...

...
```

Outro exemplo de arquivo contendo caminhos completos identificados pela sua executabilidade é apresentado a seguir. Este arquivo-exemplo contém os cinco (5) maiores caminhos completos, com restrição de duas (2) entradas em laço, que cubram elementos requeridos pelo critério *todos potenciais-usos* para a função *main* do programa cal. Antes, porém, é apresentada a lista dos elementos requeridos que originaram este arquivo-exemplo.

```
9) <1,(14,16),{ argc, argv, dayw, smon, string, mon }>
15) <1,(20,19),{ argc, argv, dayw, smon, mon }>
```

```
16) <1,(19,20),{ argc, argv, dayw, smon, string, mon }>
18) <1,(1,2),{ argc, argv, dayw, smon, string, mon }>
78) <14,(14,16),{ y }>
79) <14,(17,25),{ y }>


Caminhos Completos Gerados de acordo com a opcao:
pkteste -dmain -pu -all -e1 -k -m -c5 -r2
        -omain/pkpaths/200199/geracao/pum1k52.tes

#####################################################
numero do elemento requerido: 9

1 2 14 16 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 24 17 18 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 23 22 24 17 18 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 20 19 21 22 23 22 23 22 24 17 25
peso=11.00; tamanho=20; pred=11.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: 15

1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 21 22 24 17 1
8 19 21 22 24 17 18 19 21 22 24 17 25   --NEX
peso=21.00; tamanho=40; pred=21.00
1 2 14 16 17 18 19 21 22 24 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 1
8 19 21 22 24 17 18 19 21 22 24 17 25   --NEX
peso=21.00; tamanho=40; pred=21.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 21 22 24 17 1
8 19 20 19 21 22 23 22 24 17 25   --NEX
peso=20.00; tamanho=38; pred=20.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 2
4 17 18 19 20 19 21 22 24 17 25   --NEX
peso=20.00; tamanho=38; pred=20.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 20 19 21 22 2
4 17 18 19 21 22 23 22 24 17 25   --NEX
peso=20.00; tamanho=38; pred=20.00
numero de caminhos gerados para o elemento: 5
```

```
#####################################################
numero do elemento requerido: 16

1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 21 22 24 17 1
8 19 21 22 24 17 18 19 21 22 24 17 25   --NEX
peso=21.00; tamanho=40; pred=21.00
1 2 14 16 17 18 19 21 22 24 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 1
8 19 21 22 24 17 18 19 21 22 24 17 25   --NEX
peso=21.00; tamanho=40; pred=21.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 21 22 24 17 1
8 19 20 19 21 22 23 22 24 17 25   --NEX
peso=20.00; tamanho=38; pred=20.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 2
4 17 18 19 20 19 21 22 24 17 25   --NEX
peso=20.00; tamanho=38; pred=20.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 21 22 24 17 18 19 20 19 20 19 21 22 2
4 17 18 19 21 22 23 22 24 17 25   --NEX
peso=20.00; tamanho=38; pred=20.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: 18

1 2 14 16 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 24 17 18 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 23 22 24 17 18 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 20 19 21 22 23 22 23 22 24 17 25
peso=11.00; tamanho=20; pred=11.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: 78

1 2 14 16 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 24 17 18 19 21 22 23 22 24 17 25   --NEX
```

```
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 23 22 24 17 18 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 20 19 21 22 23 22 23 22 24 17 25
peso=11.00; tamanho=20; pred=11.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: 79

1 2 14 16 17 18 19 20 19 21 22 23 22 24 17 18 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 21 22 24 17 18 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 21 22 23 22 24 17 18 19 20 19 21 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 21 22 24 17 18 19 20 19 21 22 23 22 24 17 25   --NEX
peso=12.00; tamanho=22; pred=12.00
1 2 14 16 17 18 19 20 19 20 19 21 22 23 22 23 22 24 17 25
peso=11.00; tamanho=20; pred=11.00
numero de caminhos gerados para o elemento: 5

#####################################################
numero do elemento requerido: ...

...
```

Apêndice B

Programas da Aplicação Analisando o Número de Caminhos Executáveis

Neste apêndice são apresentadas as rotinas utilizadas no experimento apresentado no Capítulo 5, seguidas de tabelas com dados de medições refentes à sua complexidade psicológica. As métricas utilizadas para estas medições são: LOC [12, 45] , Contagem de Tokens [19], Complexidade Ciclomática ou de McCabe (V(G)) [35] e Níveis de Aninhamento [13].

Na contagem de linhas de código (LOC), considerou-se as linhas de comandos executáveis e comandos declarativos, excluindo-se as linhas de comentários e linhas em branco. Para a Contagem de Tokens utilizou-se uma adapatação, a partir da linguagem Pascal, da contagem apresentada por Conte et al.[12] para a linguagem C.

Os Níveis de Aninhamento (L0, L1, L2, L3, L4 e L5) foram contados como proposto por Dunsmore e Gannon [13], ou seja, por conjuntos de comandos agrupados aninhados sob um comando de fluxo de controle de aninhamento. Em L0 foi atribuído o número de conjuntos de comandos aninhados no nível 0 de aninhamento (nível da definição do procedimento), a L1 foi atribuído o número de conjuntos de comandos agrupados no nível 1 de aninhamento, e assim por diante.

Como o contexto em que está inserida a função no programa não é relevante para este experimento, os LOC's, tokens e aninhamentos, referentes às definições globais não são inseridos no total destas tabelas. Por este mesmo motivo, para a contagem dos tokens, as variáveis e constantes locais com o mesmo nome em diferentes procedimentos **não** são contadas como múltiplas ocorrências do mesmo operando, como recomendado em [12].

B.1 cal

```
#include <time.h>
#include <stdio.h>
```

```
char dayw[] =
{
  " S  M Tu  W Th  F  S"
};

char *smon[] =
{
  "January", "February", "March", "April",
  "May", "June", "July", "August",
  "September", "October", "November", "December"};

char string[432];

char mon[] =
{
  0,
  31, 29, 31, 30,
  31, 30, 31, 31,
  30, 31, 30, 31};

main (int argc, char *argv[])
{
  register y, i, j;
  int m;

  if (argc == 2)
    goto xlong;
  /*
   * print out just month
   */
  if (argc < 2)
    { /* current month */
      time_t t;
      struct tm *tm;

      t = time (0);
      tm = localtime (&t);
      m = tm->tm_mon + 1;
      y = tm->tm_year + 1900;
    }
  else
    {
      m = atoi (argv[1]);
      if (m < 1 || m > 12)
```

```
{
  fprintf (stderr, "cal: %s: Bad month.\n", argv[1]);
  exit (1);
}
      y = atoi (argv[2]);
      if (y < 1 || y > 9999)
{
  fprintf (stderr, "cal: %s: Bad year.\n", argv[2]);
  exit (2);
}
    }
  printf ("   %s %u\n", smon[m - 1], y);
  printf ("%s\n", dayw);
  cal (m, y, string, 24);
  for (i = 0; i < 6 * 24; i += 24)
    pstr (string + i, 24);
  exit (0);

xlong:
  /*
   * print out complete year
   */
  y = atoi (argv[1]);
  if (y < 1 || y > 9999)
    {
      fprintf (stderr, "cal: %s: Bad year.\n", argv[1]);
      exit (2);
    }
  printf ("\n\n\n");
  printf (" %u\n", y);
  printf ("\n");
  for (i = 0; i < 12; i += 3)
    {
      for (j = 0; j < 6 * 72; j++)
string[j] = '\0';
      printf ("  %.3s", smon[i]);
      printf (" %.3s", smon[i + 1]);
      printf ("        %.3s\n", smon[i + 2]);
      printf ("%s   %s   %s\n", dayw, dayw, dayw);
      cal (i + 1, y, string, 72);
      cal (i + 2, y, string + 23, 72);
      cal (i + 3, y, string + 46, 72);
      for (j = 0; j < 6 * 72; j += 72)
pstr (string + j, 72);
```

```
    }
  printf ("\n\n\n");
  exit (0);
}

pstr (char *str, int n)
{
  register i;
  register char *s;

  s = str;
  i = n;
  while (i--)
    if (*s++ == '\0')
      s[-1] = ' ';
  i = n + 1;
  while (i--)
    if (*--s != ' ')
      break;
  s[1] = '\0';
  printf ("%s\n", str);
}

cal (int m, int y, char *p, int w)
{
  register d, i;
  register char *s;

  s = p;
  d = jan1 (y);
  mon[2] = 29;
  mon[9] = 30;

  switch ((jan1 (y + 1) + 7 - d) % 7)
    {

      /*
       *    non-leap year
       */
    case 1:
      mon[2] = 28;
      break;

      /*
```

```
       *    1752
       */
    default:
      mon[9] = 19;
      break;

      /*
       *    leap year
       */
    case 2:
      ;
    }
  for (i = 1; i < m; i++)
    d += mon[i];
  d %= 7;
  s += 3 * d;
  for (i = 1; i <= mon[m]; i++)
    {
      if (i == 3 && mon[m] == 19)
{
  i += 11;
  mon[m] += 11;
}
      if (i > 9)
*s = i / 10 + '0';
      s++;
      *s++ = i % 10 + '0';
      s++;
      if (++d == 7)
{
  d = 0;
  s = p + w;
  p = s;
}
    }
}

/*
 *    return day of the week
 *      of jan 1 of given year
 */

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)
    {
      d -= (y - 1701) / 100;
      d += (y - 1601) / 400;
    }

/*
 *    great calendar changeover instant
 */

  if (y > 1752)
    d += 3;

  return (d % 7);
}
```

Tabela B.1: Medições referentes à complexidade psicológica do programa cal.c

	LOC	Contagem de Tokens				V(G)	Nível de Aninhamento				
		n1	N1	n2	N2		L0	L1	L2	L3	L4
cal	39	23	121	26	96	8	5	3	0	0	0
jan1	13	14	41	13	30	8	2	0	0	0	0
main	56	34	250	36	162	10	7	4	0	0	0
pstr	16	20	52	11	37	5	2	2	0	0	0
total	124	141	464	86	325	31	16	9	0	0	0

B.2 comm

```
#include <stdio.h>
#include "defs.h"

int one;
int two;
int three;

char ldr[3][LB];

FILE *ib1;
FILE *ib2;
/* FILE *openfil(); */

main (int argc, char *argv[])
{
  int l;
  char lb1[LB], lb2[LB];

  /* ldr[0] = "";
     ldr[1] = "\t";
     ldr[2] = "\t\t"; */

  ldr[0][0] = '\0';
  ldr[0][1] = '\0';
  ldr[1][0] = '\t';
  ldr[1][1] = '\0';
  ldr[2][0] = '\t';
  ldr[2][1] = '\t';
  ldr[2][2] = '\0';

  if (argc > 1)
    {
      if (*argv[1] == '-' && argv[1][1] != 0)
{
  l = 1;
  while (*++argv[1])
    {
      switch (*argv[1])
{
case '1':
  if (!one)
    {
```

```
      one = 1;
      ldr[1][0] = '\0';
      ldr[2][1--] = '\0';
    }
  break;
case '2':
  if (!two)
    {
      two = 1;
      ldr[2][1--] = '\0';
    }
  break;
case '3':
  three = 1;
  break;
default:
  fprintf (stderr, "comm: illegal flag\n");
  exit (1);
}
    }
  argv++;
  argc--;
}
      38
    }

  if (argc < 3)
    {
      fprintf (stderr, "comm: arg count\n");
      exit (1);
    }

  ib1 = openfil (argv[1]);
  ib2 = openfil (argv[2]);

  if (rd (ib1, lb1) < 0)
    {
      if (rd (ib2, lb2) < 0)
exit (0);
      copy (ib2, lb2, 2);
    }
  if (rd (ib2, lb2) < 0)
    copy (ib1, lb1, 1);
```

```
  while (1)
    {
      switch (compare (lb1, lb2))
{
case 0:
  wr (lb1, 3);
  if (rd (ib1, lb1) < 0)
    {
      if (rd (ib2, lb2) < 0)
exit (0);
      copy (ib2, lb2, 2);
    }
  if (rd (ib2, lb2) < 0)
    copy (ib1, lb1, 1);
  continue;
case 1:
  wr (lb1, 1);
  if (rd (ib1, lb1) < 0)
    copy (ib2, lb2, 2);
  continue;
case 2:
  wr (lb2, 2);
  if (rd (ib2, lb2) < 0)
    copy (ib1, lb1, 1);
  continue;
}
    }
  70
}

rd (FILE * file, char *buf)
{
  register int i, c;
  i = 0;
  while ((c = getc (file)) != EOF)
    {
      *buf = c;
      if (c == '\n' || i > LB - 2)
{
  *buf = '\0';
  return (0);
}
      i++;
      buf++;
```

```
    }
  return (-1);
}

wr (char *str, int n)
     /* char *str; int n; */
{
  switch (n)
    {
    case 1:
      if (one)
return;
      break;
    case 2:
      if (two)
return;
      break;
    case 3:
      if (three)
return;
    }
  printf ("%s%s\n", ldr[n - 1], str);
}

copy (FILE * ibuf, char *lbuf, int n)
{
  do
    {
      wr (lbuf, n);
    }
  while (rd (ibuf, lbuf) >= 0);

  exit (0);
}

compare (char *a, char *b)
{
  register char *ra, *rb;
  ra = --a;
  rb = --b;
  while (*++ra == *++rb)
    if (*ra == '\0')
      return (0);
  if (*ra < *rb)
```

```
    return (1);
  return (2);
}
FILE *
openfil (char *s)
{
  FILE *b;
  if (s[0] == '-' && s[1] == 0)
    b = stdin;
  else if ((b = fopen (s, "r")) == NULL)
    {
      perror (s);
      exit (1);
    }
  return (b);
}
```

Tabela B.2: Medições referentes à complexidade psicológica do programa comm.c

	LOC	Contagem de Tokens				V(G)	Nível de Aninhamento				
		n1	N1	n2	N2		L0	L1	L2	L3	L4
compare	10	14	41	13	28	4	2	1	0	0	0
copy	7	12	19	7	10	2	1	0	0	0	0
main	70	30	257	26	127	22	5	5	5	5	2
openfil	11	15	31	9	18	3	2	1	0	0	0
rd	15	19	43	15	27	3	1	1	0	0	0
wr	14	17	41	11	14	6	3	3	0	0	0
total	127	117	432	81	224	40	14	11	5	5	2

B.3 entab

```
#include "cte.h"

void settabs (tabstops)
int *tabstops;
/* Retornar 1 se i e' um
   tabstop, isso e' as colunas 0,4,8... */
```

```
{
  int i;
  for (i = 0; i < MAXLINE; i++)
    if ((i % TABSPACE) == 0)
      tabstops[i] = 1;
    else
      tabstops[i] = 0;
}

int tabpos (col, tabstops)
int col;
int *tabstops;
/* Retornar 1 se col e' em tabstop */
{
  if (col >= MAXLINE)
    return 1;
  else
    return (tabstops[col]);
}

void entab ()
/*  Substitui strings de brancos por tabs. Produz visualmente a
   mesma saida, mas com menos caracteres                          */
{
  int c, col, newcol;
  int tabstops[MAXLINE];
  settabs (tabstops);
  col = 0;
  do
    {
      newcol = col;
      while ((c = getchar ()) == BLANK)
{
  newcol++;
  if (tabpos (newcol, tabstops))
    {
      putchar (TAB);
      col = newcol;
    }
}
      while (col < newcol)
{
  putchar (BLANK);
  col++;
```

```
}
      if (c != ENDFILE)
{
  putchar (c);
  if (c == NEWLINE)
    col = 0;
  else
    col++;
}
    }
  while (c != ENDFILE);
}

void main ()
{
  entab ();
}
```

Tabela B.3: Medições referentes à complexidade psicológica do programa entab.c

	LOC	Contagem de Tokens				V(G)	Nível de Aninhamento				
		n1	N1	n2	N2		L0	L1	L2	L3	L4
entab	33	16	63	16	45	7	1	3	3	0	0
main	0	4	5	2	2	0	0	0	0	0	0
settabs	10	13	26	8	18	3	1	2	0	0	0
tabpos	9	9	16	6	10	2	2	0	0	0	0
total	52	42	110	32	75	12	4	5	3	0	0

B.4 expand

```
void expand ()
/* Obtem de uma entrada padronizada por compress,
   uma entrada normal */
{
  int c, n;
  while ((c = getchar ()) != ENDFILE)
      if (c != WARNING)
          putchar (c);
      else if (isupper (c = getchar ())){
          n = c - 'A' + 1;
```

```
            if ((c = getchar ()) != ENDFILE){
                  for (n = n; n >= 1; n--)
                  putchar (c);
            }
            else {
                putchar (WARNING);
                putchar (n - 1 + 'A');
            }
        }
        else {
            putchar (WARNING);
            if (c != ENDFILE)
                putchar (c);
        }
}

void main () {
  expand ();
}
```

Tabela B.4: Medições referentes à complexidade psicológica do programa expand.c

	LOC	Contagem de Tokens				V(G)	Nível de Aninhamento				
		n1	N1	n2	N2		L0	L1	L2	L3	L4
expand	27	17	72	7	30	7	1	2	2	3	1
main	4	4	5	2	2	0	0	0	0	0	0
total	31	21	77	9	32	7	1	2	2	3	1

As funções *main* dos programas *entab* e *expand* possuem como comando apenas a chamada dos módulos entab e expand, por isto apenas quatro linhas e nenhum nível de aninhamento.

A partir dos códigos-fonte apresentados e das tabelas B.2, B.1, B.3 e B.4, percebe-se que o programa *comm* possui maior número de sub-rotinas do que os demais programas: *cal* possui 4 sub-rotinas, *comm* possui 6, *entab* possui 4 e *expand* possui 2 sub-rotinas. Por possuir maior número de sub-rotinas e pela alta complexidade de sua função *main*, o programa *comm* apresenta maior complexidade psicológica total para as medições LOC, V(G) e quantidade de níveis de aninhamento (N). O programa *cal*, que possui a segunda maior medição de LOC e N, apresenta maior número de operandos e operadores, de acordo com a contagem de tokens, em todas as unidades n1, N1, n2 e N2, inclusive em relação ao programa *comm*. O programa *entab* apresenta-se como a terceira rotina em nível de complexidade, sendo o programa *expand* o mais simples para as medições realizadas.

Apêndice C

Resultados da Aplicação Analisando o Número de Caminhos Executáveis

Considerando o experimento apresentado no Capítulo 5, apresenta-se neste apêndice alguns resultados referentes às Etapas 3 e 4 da metodologia descrita na Seção 5.1.1.

C.1 Contagem por Elementos Requeridos

As tabelas a seguir são resultado da atividade de contagem da executabilidade de caminhos gerados pelo módulo Poke-Paths, da etapa 4 apresentada na Seção 5.1.1.

Estas tabelas apresentam a contagem de caminhos executáveis (Exec) e não executáveis (Nexe), para as estratégias menor número de predicados (Menor), maior número de predicados (Maior) e aleatória (Aletorio), por elemento requerido.

Considerando a partir da segunda linha de uma das tabelas apresentadas a seguir: a primeira coluna apresenta o número do elemento requerido (Elem); a segunda e terceira coluna apresentam respectivamente o número de caminhos executáveis (Exec) e não executáveis (Nexe) para a estratégia menor número de predicados (Menor, na primeira linha); a quarta e quinta colunas apresentam respectivamente o número de caminhos executáveis (Exec) e não executáveis (Nexe) para a estratégia maior número de predicados (Maior, na primeira linha); a sexta e sétima colunas apresentam respectivamente o número de caminhos executáveis (Exec) e não executáveis (Nexe) para a estratégia aleatória de geração de caminhos (Aleatorio, na primeira linha); e a oitava e última coluna apresenta a estratégia que gerou maior número de caminhos executáveis para aquele elemento requerido (esta coluna apresentará os termos "Menor", "Maior" e/ou "Aleatorio" referente às estratégias de geração de caminhos).

A próxima tabela apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *cal* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada é do tipo (i) (este critério é apresentado na Seção 5.1.1).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	5	0	0	5	3	2	Menor		
2	5	0	0	5	3	2	Menor		
5	3	2	0	5	1	4	Menor		
6	4	1	0	5	1	4	Menor		
7	5	0	0	5	4	1	Menor		
8	5	0	0	5	0	5	Menor		
9	3	2	0	5	2	3	Menor		
10	3	2	0	5	0	5	Menor		
11	3	2	0	5	0	5	Menor		
12	3	2	0	5	1	4	Menor		
13	3	2	0	5	0	5	Menor		
14	3	2	0	5	0	5	Menor		
15	3	2	0	5	0	5	Menor		
16	3	2	0	5	1	4	Menor		
18	0	5	0	5	0	5	Menor	Maior	Aleatorio
20	0	5	0	5	0	5	Menor	Maior	Aleatorio
21	2	3	0	5	1	4	Menor		
22	5	0	0	5	4	1	Menor		
23	5	0	0	5	4	1	Menor		
24	5	0	0	5	1	4	Menor		
25	5	0	0	5	3	2	Menor		
26	5	0	0	5	1	4	Menor		
27	5	0	0	5	3	2	Menor		
28	5	0	0	5	3	2	Menor		
29	5	0	0	5	1	4	Menor		
30	3	2	0	5	0	5	Menor		
31	3	2	0	5	1	4	Menor		
32	3	2	0	5	0	5	Menor		
33	0	5	0	5	0	5	Menor	Maior	Aleatorio
34	3	2	0	5	1	4	Menor		
36	5	0	0	5	3	2	Menor		
37	5	0	0	5	1	4	Menor		
38	5	0	0	5	3	2	Menor		
39	5	0	0	5	2	3	Menor		
40	3	2	0	5	1	4	Menor		
41	3	2	0	5	1	4	Menor		
42	3	2	0	5	2	3	Menor		
43	0	5	0	5	0	5	Menor	Maior	Aleatorio
44	3	2	0	5	1	4	Menor		

45	0	5	0	5	0	5	Menor	Maior	Aleatorio
46	5	0	0	5	4	1	Menor		
47	5	0	0	5	1	4	Menor		
48	5	0	0	5	2	3	Menor		
49	5	0	0	5	3	2	Menor		
50	5	0	0	5	3	2	Menor		
51	5	0	0	5	3	2	Menor		
53	3	2	0	5	3	2	Menor	Aleatorio	
54	3	2	0	5	1	4	Menor		
55	3	2	0	5	2	3	Menor		
56	4	1	0	5	2	3	Menor		
57	5	0	0	5	2	3	Menor		
58	4	1	0	5	2	3	Menor		
65	0	5	0	5	0	5	Menor	Maior	Aleatorio
66	0	5	0	5	0	5	Menor	Maior	Aleatorio
67	0	5	0	5	0	5	Menor	Maior	Aleatorio
71	0	5	0	5	0	5	Menor	Maior	Aleatorio
72	3	2	0	5	1	4	Menor		
73	3	2	0	5	0	5	Menor		
74	3	2	0	5	1	4	Menor		
75	0	5	0	5	0	5	Menor	Maior	Aleatorio
76	0	5	0	5	0	5	Menor	Maior	Aleatorio
77	0	5	0	5	0	5	Menor	Maior	Aleatorio
78	3	2	0	5	0	5	Menor		
79	3	2	0	5	0	5	Menor		
80	3	2	0	5	0	5	Menor		
81	3	2	0	5	0	5	Menor		
82	3	2	0	5	0	5	Menor		
83	3	2	0	5	0	5	Menor		
84	0	5	0	5	0	5	Menor	Maior	Aleatorio
85	0	5	0	5	0	5	Menor	Maior	Aleatorio
86	0	5	0	5	0	5	Menor	Maior	Aleatorio
88	3	2	0	5	1	4	Menor		
89	3	2	0	5	1	4	Menor		
90	3	2	0	5	1	4	Menor		
91	3	2	0	5	0	5	Menor		
92	3	2	0	5	1	4	Menor		
93	0	5	0	5	0	5	Menor	Maior	Aleatorio
94	3	2	0	5	0	5	Menor		
95	0	5	0	5	0	5	Menor	Maior	Aleatorio
tot:	234	161	0	395	87	308	Menor		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *cal* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	5	0	0	5	3	2	Menor		
2	5	0	0	5	3	2	Menor		
3	5	0	0	5	3	2	Menor		
4	5	0	0	5	2	3	Menor		
5	5	0	0	5	2	3	Menor		
6	0	5	0	5	0	5	Menor	Maior	Aleatorio
7	4	1	0	5	0	5	Menor		
8	0	5	0	5	0	5	Menor	Maior	Aleatorio
9	5	0	0	5	0	5	Menor		
10	3	2	0	5	0	5	Menor		
11	3	2	0	5	0	5	Menor		
tot:	40	15	0	55	13	42	Menor		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *jan1* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	2	0	2	0	2	0	Menor	Maior	Aleatorio
2	1	0	1	0	1	0	Menor	Maior	Aleatorio
3	1	1	1	1	1	1	Menor	Maior	Aleatorio
4	1	0	1	0	1	0	Menor	Maior	Aleatorio
5	2	0	2	0	2	0	Menor	Maior	Aleatorio
6	1	1	1	1	1	1	Menor	Maior	Aleatorio
8	1	0	1	0	1	0	Menor	Maior	Aleatorio
9	2	0	2	0	2	0	Menor	Maior	Aleatorio
tot:	11	2	11	2	11	2	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *jan1* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

Menor	Maior	Aleatorio

Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	1	1	1	1	1	1	Menor	Maior	Aleatorio
2	2	0	2	0	2	0	Menor	Maior	Aleatorio
3	2	0	2	0	2	0	Menor	Maior	Aleatorio
4	1	1	1	1	1	1	Menor	Maior	Aleatorio
tot:	6	2	6	2	6	2	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *main* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia é do tipo (ii) (este critério é apresentado na Seção 5.1.1).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
15	2	3	0	5	0	5	Menor		
16	2	3	0	5	0	5	Menor		
18	1	4	1	4	1	4	Menor	Maior	Aleatorio
78	0	5	1	4	1	4	Maior	Aleatorio	
79	0	5	1	4	0	5	Maior		
80	1	4	0	5	0	5	Menor		
81	1	4	0	5	0	5	Menor		
82	1	4	0	5	1	4	Menor	Aleatorio	
83	1	4	0	5	1	4	Menor	Aleatorio	
84	2	3	0	5	0	5	Menor		
85	2	3	0	5	0	5	Menor		
88	1	4	1	4	0	5	Menor	Maior	
91	2	3	0	5	0	5	Menor		
92	2	3	0	5	2	3	Menor	Aleatorio	
94	2	3	0	5	1	4	Menor		
95	2	3	0	5	1	4	Menor		
96	0	5	0	5	0	5	Menor	Maior	Aleatorio
97	2	3	0	5	1	4	Menor		
98	5	0	0	5	0	5	Menor		
99	5	0	0	5	0	5	Menor		
100	2	3	0	5	0	5	Menor		
101	1	4	0	5	0	5	Menor		
104	0	5	0	5	0	5	Menor	Maior	Aleatorio
105	0	5	0	5	0	5	Menor	Maior	Aleatorio
108	3	2	0	5	1	4	Menor		
109	3	2	0	5	2	3	Menor		
110	3	2	0	5	3	2	Menor	Aleatorio	
112	0	5	0	5	0	5	Menor	Maior	Aleatorio

114	3	2	0	5	1	4	Menor		
115	2	3	0	5	0	5	Menor		
117	0	5	0	5	0	5	Menor	Maior	Aleatorio
118	0	5	0	5	0	5	Menor	Maior	Aleatorio
119	0	5	0	5	0	5	Menor	Maior	Aleatorio
121	0	5	0	5	0	5	Menor	Maior	Aleatorio
tot:	54	146	7	193	27	173	Menor		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *main* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (ii).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
2	2	1	2	1	2	1	Menor	Maior	Aleatorio
3	1	0	1	0	1	0	Menor	Maior	Aleatorio
4	3	1	3	1	3	1	Menor	Maior	Aleatorio
5	1	0	1	0	1	0	Menor	Maior	Aleatorio
6	2	1	2	1	2	1	Menor	Maior	Aleatorio
7	5	0	5	0	5	0	Menor	Maior	Aleatorio
8	3	2	4	1	3	2	Maior		
9	1	0	1	0	1	0	Menor	Maior	Aleatorio
10	0	5	1	4	0	5	Maior		
11	0	5	1	4	0	5	Maior		
12	2	3	0	5	0	5	Menor		
13	1	4	0	5	1	4	Menor	Aleatorio	
tot:	23	30	22	31	23	30	Menor	Aleatorio	

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *main* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
9	0	5	0	5	0	5	Menor	Maior	Aleatorio
15	0	5	0	5	0	5	Menor	Maior	Aleatorio
16	0	5	0	5	0	5	Menor	Maior	Aleatorio
18	1	4	0	5	1	4	Menor	Aleatorio	
78	0	5	0	5	0	5	Menor	Maior	Aleatorio
79	0	5	0	5	0	5	Menor	Maior	Aleatorio

```
 80    0    5    0    5    0    5  Menor  Maior  Aleatorio
 81    0    5    0    5    0    5  Menor  Maior  Aleatorio
 82    0    5    0    5    0    5  Menor  Maior  Aleatorio
 83    0    5    0    5    0    5  Menor  Maior  Aleatorio
 84    0    5    0    5    0    5  Menor  Maior  Aleatorio
 85    0    5    0    5    0    5  Menor  Maior  Aleatorio
 88    0    5    0    5    0    5  Menor  Maior  Aleatorio
 89    0    5    0    5    0    5  Menor  Maior  Aleatorio
 90    0    5    0    5    0    5  Menor  Maior  Aleatorio
 91    0    5    0    5    0    5  Menor  Maior  Aleatorio
 92    0    5    0    5    0    5  Menor  Maior  Aleatorio
 94    0    5    0    5    0    5  Menor  Maior  Aleatorio
 95    0    5    0    5    0    5  Menor  Maior  Aleatorio
 96    0    5    0    5    0    5  Menor  Maior  Aleatorio
 97    0    5    0    5    0    5  Menor  Maior  Aleatorio
 98    0    5    0    5    0    5  Menor  Maior  Aleatorio
 99    0    5    0    5    0    5  Menor  Maior  Aleatorio
100    0    5    0    5    0    5  Menor  Maior  Aleatorio
101    0    5    0    5    0    5  Menor  Maior  Aleatorio
104    0    5    0    5    0    5  Menor  Maior  Aleatorio
105    0    5    0    5    0    5  Menor  Maior  Aleatorio
108    0    5    0    5    0    5  Menor  Maior  Aleatorio
109    0    5    0    5    0    5  Menor  Maior  Aleatorio
110    0    5    0    5    0    5  Menor  Maior  Aleatorio
112    0    5    0    5    0    5  Menor  Maior  Aleatorio
113    0    5    0    5    0    5  Menor  Maior  Aleatorio
114    0    5    0    5    0    5  Menor  Maior  Aleatorio
115    0    5    0    5    0    5  Menor  Maior  Aleatorio
116    0    5    0    5    0    5  Menor  Maior  Aleatorio
117    0    5    0    5    0    5  Menor  Maior  Aleatorio
118    0    5    0    5    0    5  Menor  Maior  Aleatorio
119    0    5    0    5    0    5  Menor  Maior  Aleatorio
120    0    5    0    5    0    5  Menor  Maior  Aleatorio
121    0    5    0    5    0    5  Menor  Maior  Aleatorio
tot:   1  199    0  200    1  199  Menor  Aleatorio
```

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *main* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

```
Menor     Maior   Aleatorio
```

Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	1	4	0	5	1	4	Menor	Aleatorio	
2	0	3	0	3	0	3	Menor	Maior	Aleatorio
3	1	0	1	0	1	0	Menor	Maior	Aleatorio
4	1	3	1	3	1	3	Menor	Maior	Aleatorio
5	1	0	1	0	1	0	Menor	Maior	Aleatorio
6	0	3	0	3	0	3	Menor	Maior	Aleatorio
7	0	5	0	5	0	5	Menor	Maior	Aleatorio
8	0	5	0	5	0	5	Menor	Maior	Aleatorio
9	1	0	1	0	1	0	Menor	Maior	Aleatorio
10	0	5	0	5	0	5	Menor	Maior	Aleatorio
11	0	5	0	5	0	5	Menor	Maior	Aleatorio
12	0	5	0	5	0	5	Menor	Maior	Aleatorio
13	0	5	0	5	0	5	Menor	Maior	Aleatorio
14	0	5	0	5	0	5	Menor	Maior	Aleatorio
tot:	5	48	4	49	5	48	Menor	Aleatorio	

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *pstr* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio	
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe
1	1	4	0	5	1	4
2	1	4	0	5	4	1
3	1	4	0	5	2	3
4	2	3	5	0	4	1
5	1	4	5	0	1	4
6	1	4	5	0	0	5
7	0	5	5	0	1	4
8	2	3	4	1	2	3
9	2	3	4	1	0	5
10	1	4	5	0	1	4
11	1	4	5	0	0	5
12	0	5	5	0	2	3
13	2	3	4	1	2	3
14	1	4	0	5	1	4
15	1	4	0	5	3	2
16	1	4	0	5	4	1
17	2	3	5	0	4	1
tot:	20	65	52	33	32	53

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *pstr* do programa *cal*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio	
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe
1	0	5	5	0	1	4
2	1	4	5	0	2	3
3	1	4	0	5	0	5
4	2	3	5	0	4	1
5	1	4	0	5	3	2
tot:	5	20	15	10	10	15

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *expand* do programa *expand*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio			
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe		
1	5	0	2	3	4	1	Menor	
2	4	1	0	5	3	2	Menor	
3	4	1	0	5	4	1	Menor	Aleatorio
4	4	1	0	5	3	2	Menor	
5	4	1	0	5	3	2	Menor	
6	5	0	0	5	0	5	Menor	
7	5	0	0	5	3	2	Menor	
8	3	2	0	5	2	3	Menor	
9	3	2	0	5	1	4	Menor	
16	3	2	0	5	2	3	Menor	
17	3	2	0	5	1	4	Menor	
18	3	2	0	5	2	3	Menor	
19	2	3	0	5	2	3	Menor	Aleatorio
20	5	0	0	5	3	2	Menor	
21	5	0	0	5	3	2	Menor	
22	2	3	0	5	2	3	Menor	Aleatorio
31	5	0	0	5	4	1	Menor	
32	5	0	1	4	4	1	Menor	
33	5	0	0	5	1	4	Menor	
34	5	0	0	5	1	4	Menor	
35	5	0	0	5	2	3	Menor	

36	4	1	0	5	1	4	Menor	
37	5	0	0	5	5	0	Menor	Aleatorio
38	5	0	0	5	5	0	Menor	Aleatorio
39	5	0	0	5	5	0	Menor	Aleatorio
40	5	0	0	5	5	0	Menor	Aleatorio
tot:	109	21	3	127	71	59	Menor	

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *expand* do programa *expand*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio			
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe		
1	5	0	2	3	4	1	Menor	
2	5	0	0	5	4	1	Menor	
3	3	2	0	5	2	3	Menor	
4	5	0	0	5	3	2	Menor	
5	2	3	0	5	2	3	Menor	Aleatorio
6	5	0	0	5	2	3	Menor	
7	5	0	0	5	2	3	Menor	
tot:	30	5	2	33	19	16	Menor	

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *compare* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	3	0	3	0	3	0	Menor	Maior	Aleatorio
2	3	0	3	0	3	0	Menor	Maior	Aleatorio
3	5	0	5	0	5	0	Menor	Maior	Aleatorio
4	5	0	5	0	5	0	Menor	Maior	Aleatorio
5	3	0	3	0	3	0	Menor	Maior	Aleatorio
tot:	19	0	19	0	19	0	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *compare* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

Menor		Maior		Aleatorio	

Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	3	0	3	0	3	0	Menor	Maior	Aleatorio
2	5	0	5	0	5	0	Menor	Maior	Aleatorio
3	3	0	3	0	3	0	Menor	Maior	Aleatorio
4	3	0	3	0	3	0	Menor	Maior	Aleatorio
tot:	14	0	14	0	14	0	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *copy* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	3	0	3	0	3	0	Menor	Maior	Aleatorio
2	5	0	5	0	5	0	Menor	Maior	Aleatorio
tot:	8	0	8	0	8	0	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *copy* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	5	0	5	0	5	0	Menor	Maior	Aleatorio
2	3	0	3	0	3	0	Menor	Maior	Aleatorio
tot:	8	0	8	0	8	0	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *main* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio		
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe	
1	2	3	5	0	2	3	Maior
2	2	3	5	0	2	3	Maior
3	2	3	5	0	1	4	Maior
4	2	3	5	0	2	3	Maior
5	0	5	5	0	1	4	Maior
6	0	5	5	0	1	4	Maior
7	0	5	5	0	0	5	Maior

8	0	5	5	0	0	5	Maior		
9	0	5	5	0	1	4	Maior		
10	0	5	5	0	1	4	Maior		
15	0	5	5	0	0	5	Maior		
16	0	5	5	0	0	5	Maior		
17	0	5	5	0	1	4	Maior		
18	0	5	5	0	0	5	Maior		
19	0	5	5	0	2	3	Maior		
20	0	5	5	0	1	4	Maior		
21	0	5	5	0	0	5	Maior		
22	0	5	5	0	1	4	Maior		
23	0	5	5	0	1	4	Maior		
24	0	5	5	0	1	4	Maior		
25	0	5	5	0	0	5	Maior		
26	0	5	5	0	0	5	Maior		
27	0	5	5	0	0	5	Maior		
28	0	5	5	0	0	5	Maior		
29	0	5	5	0	1	4	Maior		
30	0	5	5	0	0	5	Maior		
31	0	5	5	0	0	5	Maior		
32	0	5	5	0	2	3	Maior		
33	5	0	5	0	5	0	Menor	Maior	Aleatorio
34	5	0	5	0	5	0	Menor	Maior	Aleatorio
35	0	5	5	0	0	5	Maior		
36	0	5	5	0	1	4	Maior		
37	2	0	2	0	2	0	Menor	Maior	Aleatorio
38	5	0	5	0	5	0	Menor	Maior	Aleatorio
41	2	0	2	0	2	0	Menor	Maior	Aleatorio
42	5	0	5	0	5	0	Menor	Maior	Aleatorio
44	5	0	5	0	5	0	Menor	Maior	Aleatorio
45	5	0	5	0	3	2	Menor	Maior	
46	5	0	5	0	4	1	Menor	Maior	
47	5	0	5	0	5	0	Menor	Maior	Aleatorio
49	5	0	5	0	4	1	Menor	Maior	
50	5	0	5	0	1	4	Menor	Maior	
52	5	0	5	0	5	0	Menor	Maior	Aleatorio
53	5	0	5	0	2	3	Menor	Maior	
54	2	3	5	0	1	4	Maior		
55	0	5	5	0	2	3	Maior		
56	0	5	5	0	0	5	Maior		
57	0	5	5	0	0	5	Maior		

60	0	5	5	0	2	3	Maior		
61	0	5	5	0	2	3	Maior		
62	0	5	5	0	1	4	Maior		
63	0	5	5	0	0	5	Maior		
64	0	5	5	0	0	5	Maior		
65	0	5	5	0	0	5	Maior		
66	0	5	5	0	2	3	Maior		
67	0	5	5	0	0	5	Maior		
68	0	5	5	0	1	4	Maior		
69	5	0	5	0	5	0	Menor	Maior	Aleatorio
70	0	5	5	0	2	3	Maior		
71	5	0	5	0	5	0	Menor	Maior	Aleatorio
72	5	0	5	0	5	0	Menor	Maior	Aleatorio
73	5	0	5	0	2	3	Menor	Maior	
74	5	0	5	0	5	0	Menor	Maior	Aleatorio
75	5	0	5	0	3	2	Menor	Maior	
77	5	0	5	0	5	0	Menor	Maior	Aleatorio
78	5	0	5	0	4	1	Menor	Maior	
80	5	0	5	0	4	1	Menor	Maior	
81	3	2	5	0	2	3	Maior		
82	0	5	5	0	0	5	Maior		
83	0	5	5	0	0	5	Maior		
84	0	5	5	0	0	5	Maior		
87	0	5	5	0	1	4	Maior		
88	0	5	5	0	0	5	Maior		
89	0	5	5	0	1	4	Maior		
90	0	5	5	0	1	4	Maior		
91	0	5	5	0	0	5	Maior		
92	0	5	5	0	0	5	Maior		
93	0	5	5	0	1	4	Maior		
94	0	5	5	0	0	5	Maior		
95	0	5	5	0	1	4	Maior		
96	5	0	5	0	5	0	Menor	Maior	Aleatorio
97	0	5	5	0	1	4	Maior		
98	5	0	5	0	5	0	Menor	Maior	Aleatorio
99	5	0	5	0	5	0	Menor	Maior	Aleatorio
100	5	0	5	0	5	0	Menor	Maior	Aleatorio
102	5	0	5	0	5	0	Menor	Maior	Aleatorio
103	5	0	5	0	5	0	Menor	Maior	Aleatorio
104	5	0	5	0	4	1	Menor	Maior	
105	5	0	5	0	4	1	Menor	Maior	

106	5	0	5	0	3	2	Menor	Maior	
107	5	0	5	0	3	2	Menor	Maior	
108	5	0	5	0	3	2	Menor	Maior	
109	5	0	5	0	3	2	Menor	Maior	
111	3	2	5	0	0	5	Maior		
112	0	5	5	0	0	5	Maior		
113	0	5	5	0	1	4	Maior		
114	0	5	5	0	0	5	Maior		
117	0	5	5	0	0	5	Maior		
118	0	5	5	0	0	5	Maior		
119	0	5	5	0	2	3	Maior		
120	0	5	5	0	0	5	Maior		
121	0	5	5	0	1	4	Maior		
122	0	5	5	0	0	5	Maior		
123	0	5	5	0	2	3	Maior		
124	0	5	5	0	0	5	Maior		
125	0	5	5	0	2	3	Maior		
126	5	0	5	0	5	0	Menor	Maior	Aleatorio
127	0	5	5	0	2	3	Maior		
128	5	0	5	0	5	0	Menor	Maior	Aleatorio
129	5	0	5	0	5	0	Menor	Maior	Aleatorio
130	5	0	5	0	3	2	Menor	Maior	
132	5	0	5	0	5	0	Menor	Maior	Aleatorio
133	5	0	5	0	3	2	Menor	Maior	
134	5	0	5	0	3	2	Menor	Maior	
135	5	0	5	0	4	1	Menor	Maior	
137	5	0	5	0	5	0	Menor	Maior	Aleatorio
138	5	0	5	0	5	0	Menor	Maior	Aleatorio
139	5	0	5	0	4	1	Menor	Maior	
140	5	0	5	0	5	0	Menor	Maior	Aleatorio
141	2	3	5	0	0	5	Maior		
142	0	5	5	0	0	5	Maior		
143	0	5	5	0	0	5	Maior		
144	0	5	5	0	0	5	Maior		
147	0	5	5	0	2	3	Maior		
148	0	5	5	0	2	3	Maior		
149	0	5	5	0	1	4	Maior		
150	0	5	5	0	0	5	Maior		
151	0	5	5	0	0	5	Maior		
152	0	5	5	0	1	4	Maior		
153	0	5	5	0	0	5	Maior		

154	0	5	5	0	1	4	Maior		
155	0	5	5	0	0	5	Maior		
156	5	0	5	0	5	0	Menor	Maior	Aleatorio
157	0	5	5	0	1	4	Maior		
158	5	0	5	0	5	0	Menor	Maior	Aleatorio
159	5	0	5	0	5	0	Menor	Maior	Aleatorio
160	5	0	5	0	4	1	Menor	Maior	
161	5	0	5	0	5	0	Menor	Maior	Aleatorio
162	5	0	5	0	5	0	Menor	Maior	Aleatorio
163	5	0	5	0	4	1	Menor	Maior	
164	5	0	5	0	1	4	Menor	Maior	
165	5	0	5	0	4	1	Menor	Maior	
166	5	0	5	0	4	1	Menor	Maior	
167	5	0	5	0	5	0	Menor	Maior	Aleatorio
168	5	0	5	0	2	3	Menor	Maior	
169	5	0	5	0	4	1	Menor	Maior	
170	2	3	5	0	1	4	Maior		
171	0	5	5	0	0	5	Maior		
172	0	5	5	0	2	3	Maior		
173	0	5	5	0	3	2	Maior		
176	0	5	5	0	0	5	Maior		
177	0	5	5	0	1	4	Maior		
178	0	5	5	0	0	5	Maior		
179	0	5	5	0	0	5	Maior		
180	0	5	5	0	1	4	Maior		
181	0	5	5	0	0	5	Maior		
182	0	5	5	0	1	4	Maior		
183	0	5	5	0	1	4	Maior		
184	0	5	5	0	2	3	Maior		
185	5	0	5	0	5	0	Menor	Maior	Aleatorio
186	0	5	5	0	1	4	Maior		
187	5	0	5	0	5	0	Menor	Maior	Aleatorio
188	5	0	5	0	5	0	Menor	Maior	Aleatorio
189	0	5	5	0	3	2	Maior		
190	0	5	5	0	1	4	Maior		
191	0	5	5	0	0	5	Maior		
194	0	5	5	0	2	3	Maior		
195	0	5	5	0	0	5	Maior		
196	0	5	5	0	0	5	Maior		
197	0	5	5	0	0	5	Maior		
198	0	5	5	0	0	5	Maior		

199	0	5	5	0	1	4	Maior		
200	0	5	5	0	2	3	Maior		
201	0	5	5	0	1	4	Maior		
202	0	5	5	0	1	4	Maior		
203	5	0	5	0	5	0	Menor	Maior	Aleatorio
204	0	5	5	0	0	5	Maior		
205	5	0	5	0	5	0	Menor	Maior	Aleatorio
206	0	5	5	0	1	4	Maior		
221	5	0	5	0	5	0	Menor	Maior	Aleatorio
236	0	5	5	0	1	4	Maior		
239	0	5	5	0	0	5	Maior		
240	0	5	5	0	0	5	Maior		
241	0	5	5	0	1	4	Maior		
242	0	5	5	0	0	5	Maior		
243	0	5	5	0	0	5	Maior		
244	0	5	5	0	1	4	Maior		
245	0	5	5	0	0	5	Maior		
246	0	5	5	0	2	3	Maior		
247	0	5	5	0	1	4	Maior		
248	5	0	5	0	5	0	Menor	Maior	Aleatorio
249	0	5	5	0	1	4	Maior		
263	0	5	5	0	0	5	Maior		
264	0	5	5	0	3	2	Maior		
265	0	5	5	0	0	5	Maior		
266	0	5	5	0	0	5	Maior		
267	0	5	5	0	0	5	Maior		
268	0	5	5	0	0	5	Maior		
269	0	5	5	0	0	5	Maior		
271	0	5	0	5	0	5	Menor	Maior	Aleatorio
272	0	5	5	0	0	5	Maior		
273	5	0	5	0	5	0	Menor	Maior	Aleatorio
274	0	5	5	0	0	5	Maior		
275	0	5	5	0	1	4	Maior		
276	0	5	5	0	1	4	Maior		
279	0	5	5	0	1	4	Maior		
280	0	5	5	0	0	5	Maior		
281	0	5	5	0	1	4	Maior		
282	0	5	5	0	0	5	Maior		
283	0	5	5	0	0	5	Maior		
284	0	5	5	0	1	4	Maior		
285	5	0	5	0	4	1	Menor	Maior	

286	0	5	5	0	0	5	Maior		
287	5	0	5	0	5	0	Menor	Maior	Aleatorio
288	0	5	5	0	2	3	Maior		
301	5	0	5	0	5	0	Menor	Maior	Aleatorio
316	0	5	5	0	0	5	Maior		
319	0	5	5	0	0	5	Maior		
320	0	5	5	0	1	4	Maior		
321	0	5	5	0	2	3	Maior		
322	0	5	5	0	0	5	Maior		
323	0	5	5	0	1	4	Maior		
324	0	5	5	0	2	3	Maior		
325	0	5	5	0	1	4	Maior		
326	0	5	5	0	1	4	Maior		
328	5	0	5	0	5	0	Menor	Maior	Aleatorio
329	0	5	5	0	1	4	Maior		
343	0	5	5	0	1	4	Maior		
346	0	5	5	0	0	5	Maior		
347	0	5	5	0	0	5	Maior		
348	0	5	5	0	0	5	Maior		
349	0	5	5	0	1	4	Maior		
350	0	5	5	0	1	4	Maior		
351	0	5	5	0	3	2	Maior		
352	0	5	5	0	1	4	Maior		
353	0	5	5	0	0	5	Maior		
354	0	5	5	0	1	4	Maior		
355	5	0	5	0	5	0	Menor	Maior	Aleatorio
356	0	5	5	0	0	5	Maior		
370	0	5	5	0	2	3	Maior		
373	0	5	5	0	1	4	Maior		
374	0	5	5	0	1	4	Maior		
375	0	5	5	0	0	5	Maior		
376	0	5	5	0	2	3	Maior		
377	0	5	5	0	0	5	Maior		
378	0	5	5	0	0	5	Maior		
379	0	5	5	0	1	4	Maior		
380	0	5	5	0	1	4	Maior		
381	0	5	5	0	0	5	Maior		
382	5	0	5	0	5	0	Menor	Maior	Aleatorio
383	0	5	5	0	1	4	Maior		
tot:	384	865	1244	5	444	805	Maior		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gera-

dos para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *main* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	2	3	5	0	2	3	Maior		
2	2	3	5	0	3	2	Maior		
3	5	0	5	0	3	2	Menor	Maior	
4	5	0	5	0	4	1	Menor	Maior	
5	5	0	5	0	5	0	Menor	Maior	Aleatorio
6	5	0	5	0	4	1	Menor	Maior	
7	5	0	5	0	4	1	Menor	Maior	
8	5	0	5	0	5	0	Menor	Maior	Aleatorio
9	5	0	5	0	5	0	Menor	Maior	Aleatorio
10	5	0	5	0	5	0	Menor	Maior	Aleatorio
11	2	3	5	0	1	4	Maior		
12	0	5	5	0	1	4	Maior		
13	5	0	5	0	5	0	Menor	Maior	Aleatorio
14	0	5	5	0	1	4	Maior		
15	0	5	5	0	1	4	Maior		
16	0	5	5	0	0	5	Maior		
17	0	5	0	5	0	5	Menor	Maior	Aleatorio
18	0	5	5	0	0	5	Maior		
19	0	5	5	0	2	3	Maior		
20	5	0	5	0	5	0	Menor	Maior	Aleatorio
21	0	5	5	0	0	5	Maior		
22	0	5	5	0	0	5	Maior		
23	0	5	5	0	0	5	Maior		
24	0	5	5	0	0	5	Maior		
25	0	5	5	0	2	3	Maior		
26	0	5	5	0	0	5	Maior		
27	0	5	5	0	0	5	Maior		
tot:	56	79	130	5	58	77	Maior		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *openfil* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio	
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe

2	1	0	1	0	1	0	Menor	Maior	Aleatorio
6	1	0	1	0	1	0	Menor	Maior	Aleatorio
tot:	2	0	2	0	2	0	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *openfil* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	1	0	1	0	1	0	Menor	Maior	Aleatorio
2	1	0	1	0	1	0	Menor	Maior	Aleatorio
3	1	0	1	0	1	0	Menor	Maior	Aleatorio
tot:	3	0	3	0	3	0	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *rd* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	1	0	1	0	1	0	Menor	Maior	Aleatorio
2	5	0	5	0	5	0	Menor	Maior	Aleatorio
3	5	0	5	0	5	0	Menor	Maior	Aleatorio
4	1	0	1	0	1	0	Menor	Maior	Aleatorio
5	3	0	3	0	3	0	Menor	Maior	Aleatorio
6	5	0	5	0	5	0	Menor	Maior	Aleatorio
7	5	0	5	0	5	0	Menor	Maior	Aleatorio
8	3	0	3	0	3	0	Menor	Maior	Aleatorio
9	3	0	3	0	3	0	Menor	Maior	Aleatorio
10	5	0	5	0	5	0	Menor	Maior	Aleatorio
11	3	0	3	0	3	0	Menor	Maior	Aleatorio
tot:	39	0	39	0	39	0	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *rd* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio	
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe

```
   1    3    0    3    0    3    0  Menor  Maior  Aleatorio
   2    3    0    3    0    3    0  Menor  Maior  Aleatorio
   3    5    0    5    0    5    0  Menor  Maior  Aleatorio
tot:   11    0   11    0   11    0  Menor  Maior  Aleatorio
```

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *wr* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

```
      Menor     Maior   Aleatorio
Elem Exec Nexe Exec Nexe Exec Nexe
   1    1    0    1    0    1    0  Menor  Maior  Aleatorio
tot:    1    0    1    0    1    0  Menor  Maior  Aleatorio
```

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *wr* do programa *comm*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

```
      Menor     Maior   Aleatorio
Elem Exec Nexe Exec Nexe Exec Nexe
   1    1    0    1    0    1    0  Menor  Maior  Aleatorio
   2    1    0    1    0    1    0  Menor  Maior  Aleatorio
   3    1    0    1    0    1    0  Menor  Maior  Aleatorio
   4    1    0    1    0    1    0  Menor  Maior  Aleatorio
   5    1    0    1    0    1    0  Menor  Maior  Aleatorio
   6    1    0    1    0    1    0  Menor  Maior  Aleatorio
   7    1    0    1    0    1    0  Menor  Maior  Aleatorio
tot:    7    0    7    0    7    0  Menor  Maior  Aleatorio
```

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *entab* do programa *entab*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

```
      Menor     Maior   Aleatorio
Elem Exec Nexe Exec Nexe Exec Nexe
   1    1    4    0    5    1    4  Menor  Aleatorio
   3    1    4    0    5    0    5  Menor
   4    1    4    0    5    0    5  Menor
   5    1    4    0    5    0    5  Menor
   6    1    4    0    5    0    5  Menor
```

```
 7    1    4    0    5    0    5  Menor
 8    1    4    0    5    0    5  Menor
10    1    4    0    5    1    4  Menor  Aleatorio
11    0    5    0    5    0    5  Menor  Maior  Aleatorio
12    1    4    0    5    0    5  Menor
13    0    5    0    5    0    5  Menor  Maior  Aleatorio
14    0    5    0    5    0    5  Menor  Maior  Aleatorio
17    2    3    0    5    1    4  Menor
18    2    3    0    5    1    4  Menor
19    0    5    0    5    0    5  Menor  Maior  Aleatorio
20    1    4    0    5    1    4  Menor  Aleatorio
21    0    5    0    5    0    5  Menor  Maior  Aleatorio
22    0    5    0    5    0    5  Menor  Maior  Aleatorio
23    0    5    0    5    0    5  Menor  Maior  Aleatorio
24    1    4    0    5    0    5  Menor
25    1    4    0    5    1    4  Menor  Aleatorio
26    1    4    0    5    1    4  Menor  Aleatorio
27    2    3    0    5    0    5  Menor
28    2    3    0    5    1    4  Menor
29    1    4    0    5    1    4  Menor  Aleatorio
31    0    5    0    5    0    5  Menor  Maior  Aleatorio
33    0    5    0    5    0    5  Menor  Maior  Aleatorio
35    1    4    0    5    0    5  Menor
36    1    4    0    5    0    5  Menor
37    2    3    0    5    0    5  Menor
38    2    3    0    5    0    5  Menor
43    0    5    0    5    0    5  Menor  Maior  Aleatorio
45    0    5    0    5    0    5  Menor  Maior  Aleatorio
46    1    4    0    5    0    5  Menor
47    2    3    0    5    0    5  Menor
55    0    5    0    5    0    5  Menor  Maior  Aleatorio
57    0    5    0    5    0    5  Menor  Maior  Aleatorio
58    0    5    0    5    0    5  Menor  Maior  Aleatorio
59    1    4    0    5    0    5  Menor
62    0    5    0    5    0    5  Menor  Maior  Aleatorio
63    1    4    0    5    0    5  Menor
64    1    4    0    5    0    5  Menor
65    1    4    0    5    0    5  Menor
66    1    4    0    5    0    5  Menor
67    0    5    0    5    0    5  Menor  Maior  Aleatorio
68    1    4    0    5    0    5  Menor
```

69	0	5	0	5	0	5	Menor	Maior	Aleatorio
70	0	5	0	5	0	5	Menor	Maior	Aleatorio
72	1	4	0	5	0	5	Menor		
73	1	4	0	5	1	4	Menor	Aleatorio	
74	1	4	0	5	0	5	Menor		
75	1	4	0	5	0	5	Menor		
tot:	41	219	0	260	10	250	Menor		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *entab* do programa *entab*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	1	4	0	5	1	4	Menor	Aleatorio	
2	1	4	0	5	1	4	Menor	Aleatorio	
3	1	4	0	5	0	5	Menor		
4	2	3	0	5	0	5	Menor		
5	0	5	0	5	0	5	Menor	Maior	Aleatorio
6	0	5	0	5	0	5	Menor	Maior	Aleatorio
7	0	5	0	5	0	5	Menor	Maior	Aleatorio
tot:	6	34	0	40	4	36	Menor		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos potenciais-usos*, para a função *settabs* do programa *entab*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	0	1	0	1	0	1	Menor	Maior	Aleatorio
2	5	0	5	0	5	0	Menor	Maior	Aleatorio
3	5	0	5	0	5	0	Menor	Maior	Aleatorio
4	5	0	5	0	5	0	Menor	Maior	Aleatorio
5	5	0	5	0	5	0	Menor	Maior	Aleatorio
6	5	0	5	0	5	0	Menor	Maior	Aleatorio
7	5	0	5	0	5	0	Menor	Maior	Aleatorio
8	5	0	5	0	5	0	Menor	Maior	Aleatorio
9	5	0	5	0	5	0	Menor	Maior	Aleatorio
10	5	0	5	0	5	0	Menor	Maior	Aleatorio
tot:	45	1	45	1	45	1	Menor	Maior	Aleatorio

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *settabs* do programa *entab*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	4	1	5	0	4	1	Maior		
2	5	0	5	0	5	0	Menor	Maior	Aleatorio
3	5	0	5	0	5	0	Menor	Maior	Aleatorio
tot:	14	1	15	0	14	1	Maior		

A tabela a seguir apresenta a contagem alcançada a partir de caminhos completos gerados para a cobertura de elementos requeridos pelo critério *todos ramos*, para a função *tabpos* do programa *entab*. O critério de determinação de executabilidade adotado na etapa 4 da metodologia apresentada na Seção 5.1.1 é o do tipo (i).

	Menor		Maior		Aleatorio				
Elem	Exec	Nexe	Exec	Nexe	Exec	Nexe			
1	1	0	1	0	1	0	Menor	Maior	Aleatorio
2	1	0	1	0	1	0	Menor	Maior	Aleatorio
tot:	2	0	2	0	2	0	Menor	Maior	Aleatorio

C.2 Outras Tabelas

A seguir, são apresentadas tabelas referentes à contagem de caminhos gerados, por estratégia de seleção de caminhos, para o programa *comm*, excluída a função *main*.

As Tabelas C.1 e C.2 apresentam os resultados do programa *comm* sem a rotina main, para o critério todos-potenciais-usos e todos-ramos, respectivamente. É importante observar que, dentro do limite de laço estabelecido, foram gerados todos os caminhos possíveis por estas rotinas (*compare*, *copy*, *openfil*, *rd* e *wr*). Ainda, todos os caminhos gerados são executáveis, não provocando nenhuma diferença estatística entre as estratégias de seleção de caminhos.

Tabela C.1: Total de caminhos - critério todos-potenciais-usos (programa *comm*, excluída a função *main*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
compare	19	0	19	0	19	0	*Menor Maior Aleatória*
copy	8	0	8	0	8	0	*Menor Maior Aleatória*
openfil	2	0	2	0	2	0	*Menor Maior Aleatória*
rd	39	0	39	0	39	0	*Menor Maior Aleatória*
wr	1	0	1	0	1	0	*Menor Maior Aleatória*
total	69	0	69	0	69	0	*Menor Maior Aleatória*
total %	100,00	0,00	100,00	0,00	100,00	0,00	*Menor Maior Aleatória*

Tabela C.2: Total de caminhos - critério todos-ramos (programa *comm*, excluída a função *main*).

	Menor		*Maior*		*Aleatória*		Maior Número
	exec	Nexec	exec	Nexec	exec	Nexec	de Executáveis
compare	14	0	14	0	14	0	*Menor Maior Aleatória*
copy	8	0	8	0	8	0	*Menor Maior Aleatória*
openfil	3	0	3	0	3	0	*Menor Maior Aleatória*
rd	11	0	11	0	11	0	*Menor Maior Aleatória*
wr	7	0	7	0	7	0	*Menor Maior Aleatória*
total	43	0	43	0	43	0	*Menor Maior Aleatória*
total %	100,00	0,00	100,00	0,00	100,00	0,00	*Menor Maior Aleatória*

Apêndice D

Programas e Resultados da Avaliação de Eficácia

Este apêndice apresenta dados referentes ao experimento realizado com as estratégias de seleção de caminhos menor número de predicados, maior número de predicados e aleatória. São apresentados os defeitos inseridos nas versões incorretas do programa *cal*, dados e resultados parciais obtidos neste experimento.

O programa *cal* é utilizado no ambiente Unix e tem como função fornecer o calendário atual, do mês e/ou ano pedido. Seu código e algumas medidas de complexidade foram apresentados no Apêndice C. A seguir, são apresentadas as versões incorretas a partir do programa *cal*, seu defeito inserido no código e a linha correspondente na versão correta. Observe-se o número da versão incorreta (**X**) inserido no nome do programa (Cal-**X**.c).

```
Cal-1.c
24c24
<  if(argc = 2)
---
>  if(argc == 2)
**********
Cal-10.c
119c119
<  mon[3] = 29;
---
>  mon[2] = 29;
**********
Cal-11.c
120c120
<  mon[9] = 31;
---
>  mon[9] = 30;
```

```
***********
Cal-12.c
126a127,129
>  case 1:
>  mon[2] = 28;
>  break;
***********
Cal-13.c
149c149
<  if(i==3 || mon[m]==19) {
---
>  if(i==3 && mon[m]==19) {
***********
Cal-14.c
154a155
>  s++;
***********
Cal-15.c
146c146
<  i %= 7;
---
>  d %= 7;
***********
Cal-16.c
160c160
<  s = p;
---
>  s = p+w;
***********
Cal-17.c
181c181
<  d = 4+y+(y*3)/4;
---
>  d = 4+y+(y+3)/4;
***********
Cal-18.c
191c191
<  d += y-1601/400;
---
>  d += (y-1601)/400;
***********
```

```
Cal-19.c
199c199
<  d = 3;
---
>  d += 3;
***********
Cal-2.c
36c36
<  y = tm->tm_year + 1800;
---
>  y = tm->tm_year + 1900;
***********
Cal-20.c
77c77
<  cal(i+3, j, string+46, 72);
---
>  cal(i+3, y, string+46, 72);
***********
Cal-3.c
44c44
<  if(y<1 || y<9999) {
---
>  if(y<1 || y>9999) {
***********
Cal-4.c
52c52
<  for(i=0; i<=6*24; i+=24)
---
>  for(i=0; i<6*24; i+=24)
***********
Cal-5.c
61c61
<  if(y>9999) {
---
>  if(y<1 || y>9999) {
***********
Cal-6.c
68c68
<  for(i=0; i<12; i+=4) {
---
>  for(i=0; i<12; i+=3) {
```

```
***********
Cal-7.c
94c94
<  if(*++s == '\0')
---
>  if(*s++ == '\0')
***********
Cal-8.c
98c98
<  if(*--s == ' ')
---
>  if(*--s != ' ')
***********
Cal-9.c
108c108
<  30, 31, 30, 30,
---
>  30, 31, 30, 31,
***********
Cal.c
***********
```

O quadro abaixo apresenta as versões incorretas do programa *cal*, representadas por seus números, por tipo de erro.

Erros de Computação	*Erros de Domínio*
2, 6, 7, 9, 10, 11, 12,14, 15, 16, 17, 18, 19, 20	1, 3, 4, 5, 8, 13

Os dados de entrada utilizados neste experimento e gerados a partir dos caminhos completos são apresentados e mencionados abaixo.

1) 1 1
2) 1 4
3) 2 1
4) 3 1
5) 2 4
6) 3 4
7) 1 2
8) 2 9
9) 3 8
10) 4 1
11) 5 4
12) 1801
13) 1753
14) 1
15) 0
16) (vazio)
17) 0 1
18) 1 0

A Tabela D.1 apresenta os dados de teste encontrados para executar os caminhos completos, por estratégias que os selecionaram, para os critérios todos-potenciais-usos e todos-ramos. Estes dados de teste são representados por números que indicam a sua ordem na lista apresentada anteriormente. A Tabela D.2 apresenta os números das versões incorretas que tiveram seus defeitos revelados pelos dados de teste apresentados na Tabela D.1. As estratégias são: menor número de predicados ("Menor"), maior número de predicados ("Maior") e aleatória ("Aleatória").

Tabela D.1: Dados de entrada para os caminhos selecionados pelas estratégias (programa *cal*).

	Menor	*Maior*	*Aleatória*
todos-ramos	1, 2, 12, 13, 14, 15, 16, 17, 18	1, 12, 13, 14, 15, 16, 17, 18	1, 12, 13, 14, 15, 16, 17, 18
todos-potenciais-usos	1, 2, 3, 4, 5, 6, 12, 13, 14, 15	12, 13, 14	1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15

Tabela D.2: Versões que tiveram seus defeitos revelados pelos dados de teste da Tabela D.1.

	Menor	*Maior*	*Aleatória*
todos-ramos	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
todos-potenciais-usos	1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20	6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20	1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20