Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software

José Carlos Maldonado

Orientador: Prof. Dr. Mario Jino

Este exemplar corresponde à redação final da Dissertação apresentada à Faculdade de Engenharia Elétrica da UNICAMP, para obtenção do Título de Doutor em Engenharia Elétrica, e aprovada pela Comissão Julgadora.

Campinas, Julho de 1991

Ao meu amigo Mario Jino

Agradecimentos

Agradeço imensamente ao meu orientador, Prof. Dr. Mario Jino, por ter sugerido o tema desta tese e ter fornecido direção e confiança ao longo de sua realização; sua atuação foi fundamental para viabilizar a obtenção dos resultados apresentados. Agradeço, acima de tudo, pelo enriquecimento pessoal decorrente do convívio com sua pessoa.

Ao amigo e companheiro de trabalho Marcos L. Chaim, sem dúvida, os meus mais sinceros agradecimentos pela seriedade, dedicação, profissionalismo e, principalmente, pela amizade que permearam sua relevante participação nas diversas atividades desta tese. Suas opiniões, críticas e sugestões foram sempre pertinentes, tendo contribuido de forma relevante na obtenção dos resultados.

À Silvia R. Vergílio, pela dedicação, sugestões, companheirismo e paciência que sempre demonstrou durante a aplicação do benchmark.

Aos demais companheiros do Grupo de Teste de Software: Mauro Carnassale, Sérgio Augusto Mota Caracas, Rubens Pontes Fonseca e Plínio de Sá Leitão Júnior pelo apoio e companheirismo constantes.

Aos amigos do Departamento de Computação e Automação Industrial da UNICAMP pela consideração com que sempre fui tratado. Em especial, aos Profs. Christiano Lira Filho e Amir Said pelas sugestões e direções para a prova apresentada no Apêndice B e aos Profs. Maurício e Fernando pelas sugestões e observações durante o exame de qualificação.

Aos amigos do Departamento de Ciências de Computação e Estatística do ICMSC-USP, pelo incentivo. Em especial, aos Profs. Jorge Alberto Achcar, Josemar Rodrigues e Mariano Martínez Espinosa pelo auxílio e sugestões na análise dos resultados da aplicação do benchmark; ao Prof. Fernão pelas críticas nos momentos oportunos e à Luisa A. Spadacini Laera pela dedicação e carinho com que se empenhou na elaboração das figuras e tabelas desta tese. Gostaria de registrar meu mais profundo agradecimento àqueles que direta ou indiretamente foram sobrecarregados com atividades didáticas e administrativas em função do meu afastamento.

Aos "distantes" amigos do Dep. Ciência de Computação da Universidade Técnica da Dinamarca, pelo carinho e pela receptividade. Em especial aos amigos Dines, Hans, Morten e Mads pelas sugestões e observações técnicas.

A Augusto Cézar Freitas de Morais e irmã que não pouparam esforços para a obtenção de material bibliográfico fundamental para o desenvolvimento desta tese.

Ao amigo Tadao Takahashi pelo apoio e confiança nos momentos mais difíceis.

Ao Prof Odelar Leite Linhares pelo incentivo e motivação desde o início de minhas atividades na área de ciência de computação.

Ao Prof. Reno pela brilhante e motivadora primeira aula em computação, nos idos anos de 1974.

Aos meus amigos e aos companheiros de viagem São Carlos - Campinas, pela amizade e companheirismo.

A toda minha família que soube suportar meus momentos mais difíceis e dar o apoio necessário e essencial. Em especial, aos meus filhos pela maturidade com que enfrentaram esse longo percurso; a vocês, Marina e Thiago, peço desculpas pelas minhas ausências.

Aos membros da banca examinadora, Prof. Dr. Carlos José Pereira de Lucena, Prof. Dr. Paulo Cesar Masiero, Prof. Dr. Maurício Ferreira Magalhães e Profa. Dra. Beatriz M. Daltrini, pelas sugestões e contribuições ao trabalho, e especialmente pelo incentivo recebido para a continuidade deste trabalho.

À CAPES, CNPq, DCA/FEE/UNICAMP e ICMSC/USP pelo suporte financeiro. Enfim, a todos que, direta ou indiretamente, colaboraram para a execução deste trabalho.

Sumário

Uma família de critérios de teste estrutural baseada em análise de fluxo de dados, denominada *Família de Critérios Potenciais Usos* é definida, com a introdução do conceito *Potencial Uso*. Essa família de critérios estabelece uma hierarquia de critérios entre os critérios todos os ramos e todos os caminhos, e ainda satisfaz o requisito mínimo de cobertura do ponto de vista de fluxo de dados, mesmo na presença de caminhos não executáveis. Mostra-se que a complexidade desses critérios, assim como a dos demais critérios baseados em análise de fluxo de dados é de ordem exponencial. São caracterizados alguns modelos básicos para automatização desses critérios com o objetivo de estabelecer um núcleo básico para a automatização de critérios de teste estrutural; investiga-se o uso do conceito de arco essencial [CHU87] no contexto de teste baseado em fluxo de dados. Os principais aspectos da especificação, projeto e implementação de uma ferramenta multilinguagem, denominada POKE-TOOL, para suporte ao teste estrutural baseado em fluxo de dados de programas, são apresentados. Os resultados da aplicação de um benchmark, com o uso da POKE-TOOL, para avaliação empírica dos critérios Potenciais Usos são discutidos. A análise dos resultados obtidos indica que, do ponto de vista prático, esses critérios demandam um baixo número de casos de teste e contribuem para demonstrar que os critérios de teste estrutural baseados em análise de fluxo de dados são exeqüíveis. Vários modelos de estimativas para previsão do número de casos de teste requeridos são analisados e são explorados alguns modelos para previsão do número de caminhos não executáveis. São também discutidos alguns aspectos de medidas de complexidade de software relacionados com as atividades de teste de software.

Abstract

Potential Uses Criteria Family (PU) — a family of data flow based structural testing criteria — is defined, introducing a new concept: *the potential use*. This criteria family establishes a hierarchy including all-edges and all-paths criteria, in addition to satisfying the minimum coverage requirements from the data flow point of view, even in the presence of unexecutable paths. It is shown that the complexity of these criteria, as well as of the other data flow based criteria, have exponencial order. Some basic models are proposed aiming at establishing a uniform mechanism to automate data flow based structural testing criteria; the essential branch concept [CHU87] is investigated. The main aspects of the specification, design and implementation of a multilanguage tool, named POKE-TOOL, for data flow based structural testing of programs are presented. Results of a benchmark conducted to empirically evaluate Potential Uses Criteria, using POKE-TOOL, are discussed. Analysis of these results points out that, in practice, a small number of test cases are required by these criteria; this contributes to show that data flow based structural testing criteria can be applied in industrial software production environments. Models to estimate the number of test cases and the number of infeasible paths are explored. Some aspects of software complexity metrics related to software testing activities are also discussed.

Conteúdo

1 Introdução **1**

1.1 Contexto em que a Tese se Insere 1

1.2 Motivação 8

1.3 Objetivos da Tese 9

1.4 Organização da Tese 10

2 Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados **12**

2.1 Teste Estrutural — Terminologia e Conceitos Básicos 14

2.2 Trabalhos Relacionados 18

2.2.1 Aspectos de Automação das Atividades de Teste Estrutural de Programas 22

2.3 Considerações Finais 23

3 Critérios Potenciais Usos: Definição e Análise de Propriedades **26**

3.1 Definição da Família de Critérios Potenciais Usos 28

3.2 Análise de Propriedades: Comparação com outros Critérios de Teste 32

3.2.1 Análise de Inclusão 32

3.2.2 Análise de Complexidade 43

3.3 Considerações Finais 53

4 Modelos de Implementação dos Critérios Potenciais Usos **58**

4.1 Modelo de Fluxo de Controle 59
4.2 Modelo de Instrumentação 67
4.3 Modelo de Fluxo de Dados 69
4.4 Modelo de Descrição dos Elementos Requeridos 74
4.4.1 Utilização de Arcos Primitivos no Contexto de Fluxo de Dados . . 74
4.4.2 Grafo(i) e Descritores 82
4.5 Considerações Finais 91

5 POKE-TOOL — Arquitetura da Ferramenta de Suporte à Aplicação dos Critérios Potenciais Usos ao Teste Estrutural de Programas 92

5.1 Arquitetura da Ferramenta POKE-TOOL 94
5.2 Aspectos do Projeto e da Implementação da POKE-TOOL 98
5.2.1 Módulo Poketool 101
5.2.2 Módulo Li 104
5.2.3 Módulo Chanomat 105
5.2.4 Módulo Gera Executável 105
5.2.5 Módulo Executa Caso de Teste 106
5.2.6 Módulo Pokernel 107
5.2.7 Módulo Avaliador 113
5.3 Aspectos de Configuração da POKE-TOOL 113
5.4 Considerações Finais 116

6 Critérios Potenciais Usos: Aplicação de um Benchmark 118

6.1 Realização e Coleta de Resultados 120
6.1.1 Estratégia adotada para a condução do Benchmark 120
6.2 Análise dos Resultados do Benchmark 125
6.3 Considerações Finais 130

7 Conclusões e Trabalhos Futuros 135

7.1 Conclusões 135

7.2 Contribuição da Tese 138

7.3 Trabalhos Futuros 140

Referências 142

A A Linguagem Intermediária (LI) 149

B Determinação do Grafo de Fluxo de Controle que Maximiza o Número de Potenciais-du-caminhos 156

C Um Exemplo Completo 169

C.1 Informações Estáticas 177

C.2 Informações Dinâmicas 209

D Síntese dos Resultados e Principais Modelos Obtidos na Análise do Benchmark 219

D.1 Síntese da Análise do Benchmark para o Critério Todos-potenciais-du-caminhos 219

D.1.1 Modelos de Regressão: número de casos de teste. 229

D.1.2 Modelos de Regressão: número de caminhos não executáveis. . . . 241

Lista de Figuras

3.1 Exemplo de Aplicação dos Critérios Potenciais Usos 33

3.2 Grafo de Fluxo de Controle do Exemplo da Figura 3.1 34

3.3 Elementos Requeridos pelos Critérios Todos-potenciais-usos e Todos-potenciais-usos/du 35

3.4 Elementos Requeridos pelos Critérios Todos-potenciais-du-caminhos . . 36

3.5 Versão Incorreta do Exemplo de Aplicação dos Critérios Potenciais Usos da Figura 3.1 37

3.6 Grafo de Fluxo de Controle do Exemplo da Figura 3.5 38

3.7 Exemplo 1 para Análise de Inclusão 42

3.8 Exemplo 2 para Análise de Inclusão 44

3.9 Exemplo 3 para Análise de Inclusão 45

3.10 Exemplo 4 para Análise de Inclusão 46

3.11 Exemplo 5 para Análise de Inclusão 47

3.12 Exemplo para Análise de Inclusão na Presença de Caminhos não Executáveis 48

3.13 Ordem Parcial entre a Família de Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados (DFCF) 49

3.14 Ordem Parcial entre a Família de Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados (DFCF) na Presença de Caminhos não Executáveis 50

3.15 Ordem Parcial entre a Família de Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados (DFCF) na Presença de Caminhos não Executáveis Considerando a Propriedade NA—No-Anomalies 51

3.16 Estrutura de Controle que Maximiza o Número de Potenciais-du-caminhos 54

3.17 Exemplo para Análise de Complexidade 55

3.18 Exemplo para Análise de Complexidade na Presença de Caminhos não Executáveis com t=2 56

3.19 Exemplo para Análise de Complexidade na Presença de Caminhos não Executáveis com t=3 57

4.1 Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comandos de Seleção. 61

4.2 Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comandos de Iteração — "while" e "repeat". 62

4.3 Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comando de Iteração "for". 63

4.4 Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comandos Sequenciais e de Desvio Incondicional. . . 64

4.5 Modelo de Fluxo de Controle e Instrumentação Considerando os Comandos de Desvio Incondicional da LI: Comandos de Seleção 65

4.6 Modelo de Fluxo de Controle e Instrumentação Considerando os Comandos de Desvios Incondicional da LI: Comandos de Iteração 66

4.7 Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Seleção. 71

4.8 Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Iteração. 72

4.9 Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos Sequenciais. 73

4.10 GFC com seus Arcos Primitivos e Herdeiros. 75

4.11 Forma Geral de um Arco e seus Dois Nós. 77

4.12 Regras para Redução de Herdeiros. 78

4.13 Exemplo para Eliminação da Regra R3. 81

4.14 Grafo(1) do Exemplo do Apêndice C 83

4.15 Grafo(5) do Exemplo do Apêndice C. 84

4.16 Automatos Correspondentes aos Descritores dos Critérios Potenciais Usos. 90

5.1 Arquitetura da Ferramenta de Teste Estrutural POKE-TOOL. 96

5.2 Projeto da POKE-TOOL 100

5.3 Hierarquia das Telas da POKE-TOOL 102

5.4 Grafo de Chamada do Módulo pokernel. 109

5.5 Modelo de Dados da Linguagem C para Obtenção do Grafodef 111

B.1 Estrutura de Controle que Maximiza o Número de Potenciais-du-caminhos 161

B.2 Esquema de Du-caminhos 162

B.3 Representação das Estruturas de Controle Básicas 163

B.4 Seqüência de t Estruturas de Controle IF-THEN-ELSE 164

B.5 Seqüência de t-1 Estruturas de Controle IF-THEN-ELSE Seguida de uma Estrutura de Iteração 165

B.6 Alternativas para Adicionar uma Estrutura de Decisão a um Grafo sem Laços 166

B.7 Alternativas para Adicionar uma Estrutura de Iteração em umaa Seqüência de t-1 Estruturas IF-THEN-ELSE 167

B.8 Grafo de Fluxo de Controle que Maximiza o Número de Potenciais-du-caminhos para Programas que Nunca Terminam ("never-ending programs"). 168

Lista de Tabelas

6.1 Variáveis de Controle Relativas ao Benchmark 121

6.2 Variáveis Respostas e Associações Requeridas Relativas ao Benchmark . 124

6.3 Valores Médios 131

6.4 Melhores Modelos Obtidos 132

6.5 Síntese dos Modelos Excluídas as Unidades Recursivas e a Unidade UNROTATE 133

6.6 Síntese de Parâmetros dos Modelos da Tabela 6.5 134

D.1 Síntese dos Modelos Obtidos Considerando o Benchmark Total 221

D.2 Síntese de Parâmetros dos Modelos da Tabela 1 222

D.3 Síntese dos Modelos Obtidos Excluída a Unidade UNROTATE 223

D.4 Síntese de Parâmetros dos Modelos da Tabela 3 224

D.5 Síntese dos Modelos Obtidos Excluídas as Unidades Recursivas 225

D.6 Síntese de Parâmetros dos Modelos da Tabela 5 226

D.7 Síntese dos Modelos Excluídas as Unidades Recursivas e a Unidade UNROTATE 227

D.8 Síntese de Parâmetros dos Modelos da Tabela 7 228

Lista de Trabalhos Correlatos

[CAR91] M. Carnassale,"'GFC — Uma Ferramenta Multilinguagem para Geração de Grafo de Programa", *Tese de Mestrado*, DCA/FEE/UNICAMP Fev. , 1991.

[CHA89] - M. L. Chaim, J. C. Maldonado e Mario Jino, "Modelando a Determinação de Potenciais Du-Caminhos Através da Analíse de Fluxo de Dados," in *Proc. III Simp. Bras. Eng. de Software*, Recife, P.E., Out. 1989, pp. 112-127.

[**CHA91a**] Chaim, M.L., Maldonado, J.C. and Jino, M., *Projeto / Implementação de uma Ferramenta de Teste de Software, Relatório Técnico* DCA/RT/007/91 - DCA/FEE/UNICAMP - 1991.

[**CHA91b**] Chaim, M.L., Maldonado, J.C. e Jino, M., *Manual de Configuração da POKE-TOOL, Relatório Técnico* DCA/RT/008/91 - DCA/FEE/UNICAMP - 1991.

[**CHA91c**] Chaim, M.L., Maldonado, J.C. e Jino, M., *Manual do Usuário da POKE-TOOL,Relatório Técnico* DCA/RT/009/91 - DCA/FEE/UNICAMP - 1991.

[**CHA91d**] Chaim, M.L., "POKE-TOOL — Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados", *Tese de Mestrado*, DCA/FEE/UNICAMP - Campinas, SP, Brasil, Abril 1991.

[**CHA91e**] Chaim, M.L., Maldonado, J.C., Jino, M., "POKE-TOOL — Uma Ferramenta para Suporte à Aplicação dos Critérios Potenciais Usos para Teste de Programas", 1991 (*submetido para publicação*).

[**MAL88a**] Maldonado, J.C., Chaim, M.L., Jino, M., "Seleção de Casos de Testes baseada nos Critérios Potenciais Usos", in *Proc. II Simpósio Brasileiro de Engenharia de Software*, Canela, RS, Brasil, Out. 1988, pp. 24-35.

[**MAL88b**] Maldonado, J.C., Chaim, M.L., Jino, M., "Resultados do Estudo de uma Família de Critérios de Teste de Programas baseada em Fluxo de Dados", *Internal Technical Report* - DCA/FEE/UNICAMP - RT/DCA-001/88 - Campinas, SP, Brasil, 1988.

[**MAL89a**] Maldonado, J.C., Chaim, M.L., Jino, M., "Arquitetura de uma Ferramenta de Teste de Apoio aos Critérios Potenciais Usos", in *Proc. XXII Congresso Nacional de Informática* São Paulo, SP, Brasil, Set. 1989.

[**MAL89b**] Maldonado, J.C., Chaim, M.L., Jino, M., "Feasible Potential Uses Criteria Analysis" *Internal Technical Report* - DCA/FEE/UNICAMP - RT/DCA-001/89 - Campinas, SP, Brasil, 1989.

[**MAL91a**] Maldonado, J.C., Chaim,M.L., Jino, M., "Potential Uses Criteria Complexity Analysis", *Notas Técnicas* - DCA/FEE/UNICAMP - TN/DCA-001/91 - Campinas, SP, Brasil, 1991.

[**MAL91b**] Maldonado,J.C., Chaim, M.L., Jino, M., "Using the Essential Branch Concept to Support Data-Flow Criteria Application", 1991 (*submetido para publicação*).

[**MAL91c**] Maldonado, J.C., Chaim, M.L., Jino, M., "Potential Uses Testing Criteria: A Step towards Bridging the Gap in the Presence of Unfeasible Paths", 1991 (*submetido para publicação*).

[**MAL91d**] Maldonado, J.C., Vergílio, S.R., Chaim, M.L.; Jino, M., "Critérios Potenciais Usos: Análise da Aplicação de um Benchmark ", 1991 (*submetido para publicação*).

[**MAL91e**] J. C. Maldonado, M. L. Chaim, S. R. Vergílio, M.Jino, "Critérios Potenciais Usos: Uma Contribuição para a Atividade de Garantia de Qualidade de Software", in *Proc. Workshop em Avaliação de Qualidade de Software*, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, Maio, 1991.

[**VER91a**] Vergílio, S.R., Maldonado, J.C., Chaim, M.L., Jino, M., "Caminhos Não Executáveis na Automação da Atividades de Teste ", 1991 (*submetido para publicação*).

[**VER91b**] Vergílio, S.R., *Tese de Mestrado*, DCA/FEE/UNICAMP - Campinas, SP, Brasil, 1991 (em preparação).

Capítulo 1

Introdução

Neste capítulo são discutidos o contexto, as motivações e a relevância do tema da Tese — Teste de Software Baseado em Fluxo de Dados — dentro da área de Engenharia de Software e os objetivos principais a serem atingidos. A organização da Tese é apresentada na última seção deste capítulo.

1.1 Contexto em que a Tese se Insere

Software tem se tornado o componente central em muitas atividades complexas desenvolvidas em nossa sociedade; como conseqüência, para sua produção, técnicas especializadas e eficientes são requeridas. A *Engenharia de Software* evoluiu em resposta a essas necessidades e aplica muito dos métodos organizacionais e procedimentais da engenharia tradicional ao desenvolvimento de produtos de software.

Engenharia de Software é uma disciplina em evolução e que está em consonância com a evolução da tecnologia de computadores e com os requisitos de novas áreas de aplicação.; várias definições são fornecidas e discutidas por Mayrhauser [MAY90]. Apesar de não se ter uma definição universalmente aceita, a definição de Engenharia de Software , proposta por Fritz Bauer em uma das primeiras conferências dedicadas a essa área, capta sua essência e seus objetivos principais: produzir software de alta qualidade e de baixo custo. Qualidade de software, segundo Pressman [PRE87], pode ser definida como adequação a: i) requisitos funcionais e de desempenho explicitamente estabelecidos; ii) padrões de desenvolvimento explicitamente documentados; e iii) características implícitas que são esperadas de todo software desenvolvido profissionalmente.

Nesse sentido, a importância do uso de disciplinas de engenharia no desenvolvimento de software é clara e fundamental e deve envolver: métodos, ferramentas e procedimentos, de forma que o processo de desenvolvimento de software possa ser controlado e ainda, fornecer ao projetista subsídios para construir software de alta qualidade, de uma maneira produtiva.

A seleção e o uso, no desenvolvimento de software, desses métodos, ferramentas e procedimentos são definidos pelos diversos paradigmas; Pressman [PRE87] sugere que os diversos paradigmas devem ser encarados como complementares e propõe formas de combiná-los procurando extrair os pontos fortes de cada um deles. Independentemente do paradigma de desenvolvimento de software, o processo de desenvolvimento de software, ainda segundo Pressman, contém três fases genéricas: *definição*, *desenvolvimento* e *manutenção*.

Na *fase de definição*, identificam-se as informações a serem processadas, as funções e desempenho desejados, as interfaces a serem estabelecidas, as restrições de projeto e o critério de validação requerido; ou seja, os requisitos chave do sistema e do software são identificados. Esta fase é composta de três passos: Análise do sistema, Planejamento do Projeto de Software e Análise dos Requisitos.

A *fase de desenvolvimento* concentra-se em estabelecer *como* os objetivos serão alcançados. Descrevem-se como a estrutura dos dados e a arquitetura do software devem ser projetadas, como os detalhes procedimentais devem ser implementados, como o projeto será traduzido (transformado) para código fonte numa linguagem de programação apropriada e como os testes serão conduzidos. Três passos distintos ocorrem nesta fase: Projeto do Software, Codificação e Teste do software.

A *fase de manutenção* concentra-se em mudanças que estão associadas com a correção de erros identificados pelo usuário, adaptações requeridas em função da evolução do ambiente de software e modificações decorrentes de mudanças nos requisitos dos usuários (Ex.: funções adicionais).

Essas fases e os passos relacionados são complementados por um conjunto de outras atividades denominado *Garantia de Qualidade de Software*. *Garantia de Qualidade de Software* é uma atividade técnica cujo objetivo é garantir que tanto o processo de desenvolvimento quanto o produto atinjam níveis de qualidade especificados. *Validação e verificação* são duas atividades centrais de garantia de qualidade de software.

A *verificação* caracteriza um conjunto de atividades que garante que o produto está sendo construído corretamente; verifica-se se o processo de desenvolvimento de software está correto e se existe adequação aos padrões de organização pré-estabelecidos. A *validação*, por sua vez, tem o objetivo de garantir que o produto correto está sendo desenvolvido; o produto na sua forma corrente é comparado com os requisitos originais do usuário. A maior parte do esforço de validação tem sido realizada no final do período de desenvolvimento, quando o produto é testado; decide-se, então, se o produto está de acordo com os requisitos pré-estabelecidos. Andriole fornece uma descrição das diversas técnicas de verificação e validação utilizadas no processo de desenvolvimento de software; ele caracteriza ainda como essas diversas técnicas têm sido integradas no ciclo de vida de desenvolvimento de software [AND86].

No processo de desenvolvimento de software todos os erros são erros humanos e, apesar da introdução de melhores métodos de desenvolvimento, melhores ferramentas de suporte e treinamento de pessoal, erros permanecem presentes nos diversos produ-

tos de software produzidos e liberados [HOW87]; nesse contexto, a atividade de teste continuará a ter um papel importante no desenvolvimento de software. Myers considera que a maioria dos erros origina-se na comunicação e transformação de informações [MYE79]. Padrões típicos da indústria de software são da ordem de 10 erros por KDSI (1000 linhas de código fonte liberadas); softwares de alta qualidade apresentam da ordem de 1 erro por KDSI, enquanto que para softwares extremamente confiáveis a taxa de erro é da ordem de 0.1 erro por KDSI. Um ponto importante é que a maioria dos erros encontra-se em partes do código raramente executadas.

Myers identifica três abordagens complementares para prevenir ou detectar erros [MYE79]:

i) introduzir maior precisão no processo de desenvolvimento de software;

ii) introduzir, em cada fase ou passo do processo de desenvolvimento de software, uma atividade de verificação; o objetivo principal é localizar o maior número possível de erros antes de passar para o próximo passo ou etapa; e

iii) orientar diferentes níveis de testes em relação às diversas fases (etapas) do processo de desenvolvimento, focalizando em classes distintas de erros.

A vantagem da abordagem de desenvolvimento modular e sistemático é que a complexidade é reduzida pela decomposição da descrição e definição dos diferentes aspectos do sistema de software. Isto simplifica a descrição e torna o desenvolvimento mais fácil e mais controlável; viabiliza-se a detecção de erros básicos no início das atividades de desenvolvimento, obtendo-se um decréscimo no custo da eliminação desses erros e, consequentemente, um decréscimo no custo global de desenvolvimento. Um exemplo de desenvolvimento de software mais rigoroso e com o uso de formalismos matemáticos é o método *VDM* [JONES86, BJO78]. Pressman fornece uma visão geral dos métodos e técnicas que têm sido introduzidos e utilizados no desenvolvimento de software [PRE87]. Do ponto de vista de teste de software, Andriole [AND86] e Gelperin e Hetzel [GEL88] ilustram os aspectos complementares das diferentes abordagens de prevenção e detecção de erros utilizadas no desenvolvimento de software.

Um dos produtos do desenvolvimento de software é a especificação dos requisitos e características desejadas do sistema. A *especificação do software* fornece (mesmo que informalmente) uma descrição dos dados de entrada do programa; esse conjunto de dados é denominado *domínio do programa* e é, usualmente, representado por D. A especificação também fornece uma descrição do comportamento esperado do programa no domínio D; o comportamento esperado para um dado de entrada $d \in D$ é usualmente representado por $f(d)$. Semelhantemente, um programa P representa algumas ações computacionais que são executadas quando dados de entrada são fornecidos; o comportamento do programa é representado simplificadamente por P^*. Desta forma, $P^*(d)$ é uma idealização matemática do comportamento do programa P tendo como entrada o item de dado d.

Diz-se que um programa P é correto com respeito a uma especificação f se $f(d) = P^*(d)$ para qualquer item de dado d pertencente a D, ou seja, se o comportamento do programa está de acordo com o comportamento esperado para todos os dados de entrada. A correção de um programa não é uma propriedade absoluta, mas sim uma propriedade relativa; o programa é considerado correto em relação a alguma descrição do que o usuário espera que o programa faça [HOW87].

No teste de programa, pressupõe-se que alguém (um testador) ou algum mecanismo — um *oráculo* — possa determinar, para qualquer item de dado $d \in D$ se $f(d) = P^*(d)$, dentro de limites de tempo e esforço razoáveis. Um *oráculo* decide simplesmente se os valores de saída são corretos e não se eles são computados corretamente; se esses valores forem incorretos o oráculo não fornece qualquer informação sobre o erro nem o valor de saída correto. Weyuker [WEY82] discute detalhadamente a importância e a sensatez de assumir-se a existência de um oráculo nas atividades de teste.

Teste pode ser visto como a atividade final de verificação e validação dentro da organização de desenvolvimento de software [JON90]; teste é uma parte fundamental em todos os ramos da engenharia e é uma parte essencial no desenvolvimento de software [HOW87]. A necessidade de testes é oriunda da inabilidade de se garantir que as demais atividades do projeto de software foram realizadas adequadamente [GEL88, SOM89].

As atividades de teste consomem, em projetos típicos de programação, da ordem de 50% do tempo e do custo de desenvolvimento de um produto de software. Apesar disto, a observação de Myers ainda é pertinente nos dias de hoje: aparentemente, conhece-se muito menos sobre teste de software do que sobre outros aspectos e/ou atividades do desenvolvimento de software [MYE79]. Outra atividade em situação semelhante é a *manutenção de software*; estimativas indicam que mais de 50% do orçamento de um produto de software é gasto com atividades de manutenção. De acordo com Gregory Jones, o custo de manutenção frequentemente atinge 70% do orçamento e algumas estimativas indicam valores de até 82% [JON90].

Usualmente, caracterizam-se três fases (níveis) de teste, usualmente presentes em uma estratégia de teste: *teste de unidade*, *teste de integração* e *teste de sistema*. *O teste de unidade* concentra esforços na menor unidade do projeto de software: o módulo; visa a identificar erros de lógica e de implementação. O *teste de integração* é uma técnica sistemática para integrar os módulos componentes na estrutura do software; visa, essencialmente, a identificar erros de interface entre os módulos. O *teste de sistema* é conduzido após a integração do sistema e visa a identificar erros de função e/ou de características de desempenho (que não estejam de acordo com a especificação).

De uma maneira geral, o teste de software, que é uma das atividades de garantia de qualidade de software, envolve as seguintes atividades: *planejamento*, *projeto de casos de teste*, *execução de casos de teste* e *análise (avaliação) dos resultados dos testes*. A ausência de planejamento das atividades de desenvolvimento é uma das origens da crise de software [PRE87]. O planejamento da atividade de teste deve

fazer parte do planejamento global do sistema, culminando num *Plano de Teste* que constitui um documento crucial no ciclo de vida de desenvolvimento de software. Nesse plano, são estimados recursos e são definidos estratégias, métodos e técnicas de teste , caracterizando-se um critério de aceitação do software em desenvolvimento; a falta de tempo e de recursos, e a indisponibilidade de ferramentas adequadas são os principais problemas enfrentados pelas equipes de teste.

Uma questão usualmente presente na elaboração de um plano de teste é saber quando as atividades de teste devem ser encerradas, ou seja, estabelecer um *critério de parada*. O que é necessário é a adoção de algumas métricas para caracterizar a qualidade do teste e também a qualidade do software [HAL91]. As atividades de teste não devem ser encerradas se existe insatisfação quanto à determinação da qualidade do software ou, se existe uma chance razoável de se detectar a presença de erros ainda não detectados. A qualidade do teste é um fator muito importante pois, se um produto de software é submetido a todos os testes e não é indicada a presença de erros, isso pode significar simplesmente que o conjunto de casos de teste é de baixa qualidade e não que o software seja de alta qualidade. Myers discute alguns critérios de parada de teste [MYE79]; um enfoque alternativo ao discutido por Myers é a análise de mutantes [COW88]. *Métricas de cobertura* de teste são um excelente método para medir a qualidade dos testes [HAL91]. Métricas de cobertura de teste são estabelecidas a partir do conhecimento de detalhes de implementação e o objetivo é garantir que os testes "cobriram" todo o código; usualmente, essas métricas estão fortemente relacionadas com critérios de teste estruturais, como será visto na seqüência desta tese.

Uma outra atividade importante da Engenharia de Software é o estabelecimento de métricas que viabilizem caracterizar, estimar e planejar o processo de desenvolvimento, bem como o próprio produto alvo. Essas métricas orientam a coleta de informação ao longo do desenvolvimento; essa informação auxilia no controle do projeto corrente e pode fornecer subsídios para o aperfeiçoamento dos modelos de planejamento. Sem o auxílio de dados históricos, estimativas resultam em previsões de baixa qualidade; sem uma sólida indicação da produtividade, não se pode avaliar a eficácia e eficiência de novas ferramentas, técnicas e padrões.

Quanto ao *projeto de casos de teste*, idealmente, o programa deveria ser exercitado para todos os valores (dados) de entrada possíveis. No entanto, sabe-se que o teste exaustivo é impraticável devido a restrições de tempo e custo. Um dos resultados teóricos mais importantes na área de teste de programas é que não existe um procedimento de teste de propósito geral que possa ser usado para provar a correção de um programa. Consequentemente, afirmações do tipo "teste de software pode ser usado somente para detectar a presença de erros, e não para provar sua ausência" têm sido feitas quando da introdução de métodos formais e de métodos de prova no desenvolvimento de software. Howden aponta várias limitações da abordagem formal e considera que o uso cooperativo e complementar dessas duas abordagens deve evoluir [HOW87].

Neste ponto, caracterizaram-se as duas questões chaves de pesquisa na área de teste de software: " Como os dados de teste devem ser selecionados?" e "Como se pode

dizer se um programa P foi testado suficientemente?". Frankl define *método de seleção de dados de teste* como um procedimento para escolher *casos de testes*; e *critério de adequação dos dados de teste* (conjunto de dados de teste) como um predicado usado para avaliar o(s) dado(s) de teste. Frankl observa ainda que existe uma correspondência entre um método de seleção de dados de teste e critérios de adequação de dados de teste. Dado um critério de adequação C, existe um método M_C de seleção de dados de teste que estabelece: "selecione um conjunto de teste que satisfaz o critério C". De forma análoga, dado um método de seleção M, existe um critério de adequação C_M que diz: "um conjunto de dados de teste é adequado se ele foi gerado pelo método M". Nesta tese usar-se-á o têrmo critério com ambos os sentidos [FRA87]. Em geral, um critério de adequação é um dos itens de critérios de parada das atividades de teste [MYE79].

Segundo Frankl [FRA87] existem várias fontes de informação disponíveis para auxiliar na seleção de dados de teste de software, entre elas: o texto do programa, a especificação do programa e informações sobre erros comuns no desenvolvimento de software. O uso dessas informações no estabelecimento de critérios de teste, ao invés de estabelecê-los de forma "ad hoc", tem em geral maior probabilidade de indicar a presença de erros existentes no programa. A distinção entre dados de entrada e casos de teste é muito importante nas atividades de teste [MYE79]; um caso de teste consiste da especificação do dado de teste (dado de entrada) e do comportamento esperado do programa P. Desta maneira, pode-se confrontar o resultado efetivamente obtido com o resultado esperado.

Uma técnica que se utiliza principalmente da especificação do software para derivar os requisitos de teste é denominada *técnica baseada em especificação* ("specification based"); é também conhecida como *técnica funcional* ou *técnica de caixa preta* [CLA82, WEY80, RIC81, RIC85]. Aquela que se utiliza principalmente de informações sobre os erros mais comuns cometidos durante o processo de desenvolvimento de software é chamada de *técnica baseada em erros* ("error based") [GOD75, WEY80, DEM78, HOW81, CLA82, OST79, WHI80]. A que se utiliza principalmente de informações oriundas do texto de programa é denominada *técnica baseada em programa* ("program based"); é também denominada *técnica estrutural de teste* ou *de caixa branca* [KIN76, HOW75, HER76, LAS83, NTA84 RAP82, HOW78, RAP85, URA88, WOO80].

A técnica baseada em erros usa informações sobre o processo de desenvolvimento de software para identificar os tipos mais comuns de erros presentes em um produto de software; em geral, procura-se caracterizar os erros em termos de seus efeitos potenciais nas características do software implementado, mais especificamente em caminhos do programa — possíveis seqüências de comandos da entrada para a saída. Os erros são classificados em dois tipos: *erros computacionais*: o erro provoca uma computação incorreta e o caminho executado é igual ao caminho esperado; e *erros de domínio*: o caminho efetivamente executado é diferente do caminho esperado, ou seja, um caminho errado é selecionado.

No contexto desta tese o termo *erro* será usado para referenciar um defeito (causa)

no programa, e o termo *falha* (sintoma) a um comportamento incorreto do programa induzido pela existência de um ou mais erros no programa.

A técnica funcional de teste aborda o software de um ponto de vista macroscópico; trata o programa como uma caixa preta, onde o conteúdo (detalhes da implementação) não é (são) conhecido(s). Apenas o comportamento esperado é conhecido, a partir da especificação funcional do software.

A técnica estrutural de teste é baseada no conhecimento da estrutura interna da implementação e o objetivo é caracterizar um conjunto de componentes elementares de um programa que devem ser exercitados ("cobertos") pelo conjunto de casos de teste; diferentes abordagens correspondem a diferentes tipos de componentes (por exemplo: comandos, seqüências de comandos, laços, etc.).

A técnica estrutural de teste (baseada no programa) é mais adequada para o teste de unidade. Uma vantagem da técnica funcional de teste é que ela é adequada em todos os níveis (fases) de teste. No entanto, segundo Gregory Jones [JON90], frequentemente uma função especificada é implementada por um conjunto de unidades, tornando o teste funcional impraticável durante o teste de unidade. Mayrhauser discute várias vantagens e desvantagens entre essas duas abordagens e considera que o ponto principal não é *escolher entre* as técnicas e sim *quando usar* uma delas [MAY90]. Independentemente das vantagens e desvantagens dessas técnicas, elas devem ser vistas como complementares, uma vez que cobrem essencialmente classes distintas de erros [MYE79, PRE87, MAY90]. Vários estudos teóricos e empíricos têm sido conduzidos e ratificam esta afirmação [DUR84, HAM88, JEN89, HOL79, GOU83]. Recentemente, vários critérios, métodos e estratégias de teste que combinam essas técnicas [HOW87, RIC81, WEY80] têm surgido.

Existem diferenças fundamentais nas definições da abrangência e dos objetivos de teste utilizados. Essas diferenças têm implicações diretas nas abordagens adotadas na condução das diversas atividades de teste (por ex. na geração de casos de teste) e implicam em diferentes visões do que é um teste bem sucedido. Gelperin identifica quatro modelos de teste que evoluíram em função da própria evolução do entendimento das atividades e do processo de teste: dois modelos de fase (modelo de demonstração e modelo de destruição) e dois modelos de ciclo de vida (modelo de avaliação e modelo de prevenção) [GEL88].

No contexto desta tese, o termo teste de software não abrange outras atividades de avaliação do software tais como prova de correção e revisões do software. Coward, por exemplo, adota um significado mais abrangente para este termo [COW88].

Duas abordagens para a definição dos objetivos do teste podem ser caracterizadas [HEL88, COW88]: teste como um *processo destrutivo*, onde o objetivo é encontrar falhas (indicativo da presença de erros); e teste como um *processo construtivo*, onde o objetivo é demonstrar que o software não tem erros.

Entende-se, no contexto desta tese, que o principal objetivo do teste de software é

revelar a presença de erros no programa em teste: em outras palavras, o objetivo do teste de software é refutar a afirmação de que o programa está correto; vários outros autores adotam esta mesma abordagem [MYE79, PRE87, HAL91, COW88]. Frankl acrescenta: "o objetivo do teste de software é detetar a presença de erros ou, não atingindo este objetivo, aumentar a confiança de que o programa está correto" [FRA87]. Coward compartilha desta visão, considerando que se os testes foram rigorosos, pode-se estabelecer uma maior confiança no software [COW88]. Nesse contexto, um *teste bem sucedido* é aquele que revela a presença de um erro; um *bom caso de teste* é aquele que tem uma alta probabilidade de encontrar um erro ainda não descoberto. Distingue-se ainda, entre teste e *depuração*; no início das atividades de desenvolvimento de software esses dois termos eram usados indistintamente. Teste consiste em indicar a presença de erros, enquanto depuração consiste no conjunto de atividades que tem por objetivo localizar, identificar e corrigir um erro.

1.2 Motivação

A partir da caracterização do contexto em que a tese se insere, podem-se extrair alguns pontos relevantes que foram determinantes para a condução deste trabalho de pesquisa.

- Produtos de software têm sido utilizados cada vez mais em atividades essenciais às atividades humanas e, em conseqüência, a necessidade de técnicas e ferramentas para a obtenção de produtos confiáveis e de baixo custo é clara;
- Erros estão presentes na maioria dos produtos de software produzidos e liberados, apesar dos avanços significativos que se têm obtido na área de Engenharia de Software;
- As atividades de teste são responsáveis pelo comprometimento de até 50% do tempo e do custo de desenvolvimento de software, em projetos típicos;
- O teste de software é uma das técnicas mais utilizadas, atualmente, para a validação de produtos de software;
- As técnicas funcional e estrutural são complementares, pois cobrem classes distintas de erros;
- O suporte automatizado aos métodos, técnicas e critérios proposto para o desenvolvimento de software é essencial, do ponto de vista de produtividade; e
- A informação das atividades de teste é fundamental para a atividade de depuração e de manutenção do software; note-se que a manutenção é a atividade que mais consome recursos de uma organização de desenvolvimento de software.

Além desses pontos, um outro fator relevante foi que, no início de desenvolvimento desta tese, no contexto brasileiro, poucos pesquisadores realizavam trabalho nesta área;

esta situação não se alterou essencialmente até os dias de hoje. A importância de introduzir rigor e sistematização, assim como suporte automatizado, em todas as atividades de desenvolvimento de software é evidente, não só socialmente, quanto para o próprio país, na medida em que os nossos produtos atinjam níveis de qualidade aceitos pela comunidade internacional e se tornem competitivos em termos de custo.

1.3 Objetivos da Tese

O interesse e direção principal de pesquisa desta tese concentram-se no estudo de técnicas de teste estrutural, mais especificamente no estudo de teste estrutural baseado em análise de fluxo de dados [HEC77].

As técnicas baseadas em análise de fluxo de dados caracterizam (agrupam) os componentes elementares do programa que têm uma relação de fluxo de dados (por exemplo, uma definição e uso de uma variável), e requerem que esses elementos sejam exercitados pelos casos de teste. Segundo Howden [HOW87] este é um enfoque promissor, pois uma interpretação possível do teste baseado em fluxo de dados é a de que comandos que têm uma relação de fluxo de dados são provavelmente partes de uma mesma função embutida e devem ser exercitados (testados) juntamente, pelo menos uma vez.

Além de estudar os princípios básicos do teste estrutural baseado em fluxo de dados, o interesse é investigar e determinar mecanismos para o suporte automatizado à aplicação dessa classe de critérios, uma vez que, sem o apoio automatizado, a sua utilização é limitada. O suporte automatizado ao uso de critérios de teste estrutural como critérios de adequação, requer facilidades tais como: instrumentação de programas, monitoração de caminhos efetivamente executados pelos casos de teste e meios para se decidir se o critério foi satisfeito ou não. Com a disponibilidade de uma ferramenta de teste e através de estudos empíricos, espera-se obter subsídios que forneçam uma indicação da viabilidade da aplicação desses critérios em programas reais ou em ambientes reais de produção de software.

Um outro objetivo é investigar o uso do conceito de *arco essencial*, dentro do contexto de teste baseado em fluxo de dados, para a descrição dos elementos requeridos e para a análise de adequação de conjuntos de casos de teste em relação a um dado critério. Uma das metas é obter um mecanismo uniforme que facilite a automatização de outros critérios, uma vez que atividades de comparação teórica e empírica entre critérios de teste de software têm predominado nos dias de hoje, almejando fornecer subsídios para a seleção de critérios. O conceito de arco essencial foi introduzido por Chusho para evitar duas deficiências do teste de ramos: a seleção de dados de teste redundantes e a superestimação da qualidade do software [CHU87]; Chusho, através de um estudo prático, observou uma redução da ordem de 40% no número de arcos a serem monitorados e avaliados, o que implica uma redução do custo total das atividades de teste.

Modelos para estimar o esforço das atividades de teste são iniciativas que contribuem para uma prática mais sistemática e disciplinada no desenvolvimento de um produto de software. A quantificação do esforço de teste pode então ser incorporada em planos de teste e auxiliar nas atividades de garantia de qualidade. Outros objetivos desta tese consistem em: i)Determinar modelos para estimar, nas diversas etapas de desenvolvimento do software, o número de casos de teste necessários para testar um dado programa com o uso de critérios baseados em fluxo de dados; e ii) Explorar a influência de outras características do produto em teste, além do número de comandos de decisão, na determinação dos modelos de estimativas. Por exemplo, seria de se esperar que o número de variáveis utilizadas no programa fosse significativo na estimativa do número de casos de teste necessários para satisfazer os critérios baseados em análise de fluxo de dados.

1.4 Organização da Tese

Neste capítulo situou-se o contexto desta tese e caracterizou-se sua relevância dentro do contexto de Engenharia de Software; foram também definidos os objetivos principais da tese.

No Capítulo 2 introduzem-se os conceitos, terminologia e limitações mais específicos ao teste estrutural e apresenta-se uma síntese de outros trabalhos relacionados.

No Capítulo 3 introduz-se a Família de Critérios Potenciais Usos e determinam-se suas propriedades teóricas analisando-se a ordem parcial, com base em uma relação de inclusão, entre esses critérios e os demais critérios baseados em fluxo de dados; a complexidade — o número máximo de casos de teste requeridos, no pior caso — dos critérios Potenciais Usos é também determinada e discutem-se alguns pontos de discordância quanto à complexidade dos demais critérios de teste baseados em fluxo de dados.

No Capítulo 4 são discutidos alguns aspectos teóricos que nortearam a especificação e implementação da ferramenta de suporte à aplicação dos Critérios Potenciais Usos. Considerando-se que uma das direções de pesquisa de interesse é a combinação de critérios, o que envolve certamente atividades de comparação teórica e empírica entre diversos critérios, quatro modelos básicos são propostos: *Modelo de Grafo de Fluxo de Controle; Modelo de Instrumentação; Modelo de Dados; e, Modelo de Descrição dos Elementos Requeridos.* Na proposição desses modelos, procurou-se abstrair alguns mecanismos que pudessem caracterizar um conjunto de recursos que facilitasse a incorporação de outros critérios na POKE-TOOL. O uso do conceito de *arco essencial (arco primitivo)* dentro do contexto de fluxo de dados é explorado; é proposta uma mudança no algoritmo de Chusho [CHU87], adequando-se esses conceitos para a descrição dos elementos requeridos pelos critérios Potenciais Usos e para a análise de adequação de conjuntos de casos de teste.

A arquitetura e os principais aspectos funcionais da ferrramenta de teste POKE-TOOL são discutidos no Capítulo 5; os modelos básicos propostos no capítulo anterior são utilizados na proposição da arquitetura desta ferramenta. A POKE-TOOL é uma ferramenta flexível, isto é, não atrelada a uma linguagem de programação específica, permitindo que o usuário a configure para a linguagem de programação de seu interesse, de acordo com um procedimento de configuração. É também apresentada uma síntese dos aspectos do projeto e da implementação, e das etapas para configuração da POKE-TOOL, extraída de [CHA91a, CHA91b].

No Capítulo 6 são descritas a realização e análise da aplicação dos Critérios Potenciais Usos, utilizando-se a POKE-TOOL, no mesmo benchmark utilizado por Weyuker [WEY90] — extraído do livro de Kernighan & Plauger [KER81]. Obtém-se como resultado da aplicação do benchmark vários modelos para estimar o número de casos de teste requeridos; duas medidas de cobertura em função do conceito *potencial-du-caminho* são propostas. Alguns aspectos quanto à não executabilidade de caminhos são também explorados.

As conclusões, contribuições da tese e proposições de trabalhos futuros estão contidas no Capítulo 7.

A tese contém quatro apêndices. O Apêndice A contém uma síntese da descrição de uma *linguagem intermediária* (LI) [CAR91, CHA91a] que visa a identificar o fluxo de controle em um programa; é a partir desta linguagem que são caracterizados alguns dos modelos discutidos no Capítulo 4. O Apêndice B consiste na determinação (prova) de uma estrutura de fluxo de controle que maximiza o número de "potenciais-du-caminhos" — conceito fundamental no estabelecimento dos critérios Potenciais Usos; essa estrutura é muito importante para a análise de complexidade dos critérios baseados em fluxo de dados. O Apêndice C ilustra alguns aspectos da operação da POKE-TOOL e contém as informações mais relevantes geradas por essa ferramenta para um programa extraído do benchmark. Finalmente, o Apêndice D apresenta informações mais detalhadas relativas à análise da aplicação do benchmark, discutida no Capítulo 6.

Capítulo 2

Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados

Primeiramente, retomamos alguns pontos fundamentais discutidos no Capítulo 1. Uma das metas da Engenharia de Software é produzir software de alta qualidade. Teste de software é uma das atividades de garantia de qualidade de software; de acordo com Chusho [CHU87] é a chave para aprimorar a produtividade e a confiabilidade do software. Teste de software envolve: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes. O projeto de casos de testes concentra-se em um conjunto de técnicas, critérios e métodos para elaborar os casos de teste. Estes métodos, critérios e técnicas fornecem ao projetista de software uma abordagem sistemática para as atividades de teste de software; além disso, constituem um mecanismo que pode auxiliar a garantir a completude dos testes e uma maior probabilidade de revelar defeitos no software. Em geral, deve-se projetar casos de testes que tenham a maior probabilidade de encontrar o número máximo de defeitos com um mínimo de tempo e esforço.

A técnica estrutural de teste de programas é baseada no conhecimento da estrutura interna da implementação do software e visa a caracterizar um conjunto de componentes elementares do software que devem ser exercitados. Informalmente, a estrutura interna é representada por um *grafo de fluxo de controle (grafo de programa)* —um grafo dirigido, com um único nó de entrada e um único nó de saída —, onde cada nó representa uma seqüência de comandos que são sempre executados como um bloco de comandos e cada arco representa uma transferência de controle entre esses blocos; um caminho de um programa é representado por uma seqüência de nós.

A técnica estrutural de teste de programas apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes às atividades de teste de programas enquanto estratégia de validação [HOW87, FRA87, NTA88, RAP85]. Esses aspectos introduzem sérias limitações na automatização do processo de validação de software:

- não existe um procedimento de teste de propósito geral que possa ser usado para provar a correção de um programa;
- dados dois programas, é indecidível se eles computam a mesma função;
- é indecidível, em geral, se dois caminhos de um programa, ou de programas diferentes, computam a mesma função; e
- é indecidível, em geral, se um dado caminho é executável, ou seja, se existe um conjunto de dados de entrada que leva à execução desse caminho.

As limitações inerentes à técnica estrutural de teste são:

- caminhos ausentes — se o programa não implementa algumas funções, não existirá um caminho que corresponda àquela função e, consequentemente, nenhum dado de teste será requerido para exercitá-la.
- correção coincidente — o programa pode apresentar, coincidentemente, um resultado correto para um item particular de dado $d \in D$; esta limitação pode ser considerada como uma limitação fundamental para qualquer estratégia de teste.

Independentemente dessas desvantagens da técnica estrutural de teste ela é vista como complementar à técnica funcional, uma vez que cobre classes distintas de erros [MYE79, PRE87, HOW87, MAY90]; alguns resultados de pesquisa apontam o desenvolvimento formal de software como uma alternativa ao uso da técnica estrutural de teste [DYE90]. No entanto, os conceitos e experiência obtidos com o teste estrutural podem ser abstraídos e incorporados à técnica funcional [HOW87, RIC81, WEY80]. Ainda, as informações obtidas com a aplicação da técnica estrutural têm sido consideradas relevantes para as atividades de manutenção e depuração de software [OST88].

Os primeiros critérios de teste estrutural de programas utilizados eram baseados unicamente no fluxo de controle de programas, sendo os mais conhecidos: o critério *todos os nós*; o critério *todos os ramos*; e o critério *todos os caminhos*. O critério *todos os nós* requer que todos os comandos sejam executados pelo menos uma vez; o segundo deles, o critério *todos os ramos*, um dos critérios mais utilizados, requer que toda transferência de controle entre blocos de comandos seja exercitada pelo menos uma vez. O critério *todos os caminhos*, por sua vez, requer que todos os caminhos possíveis do programa sejam exercitados[HOW78, WOO80, KIN76, HOW75].

Posteriormente, surgiram os critérios baseados em análise de fluxo de dados do programa. A análise de fluxo de dados [HEC77] tem sido amplamente utilizada para otimização de códigos por compiladores e para detecção de anomalias no programa, através de análise estática do programa; em geral, classifica cada ocorrência de uma variável no programa como uma definição ou como um uso. Esses critérios usam portanto, informações do fluxo de dados do programa para derivar os requisitos de teste.

Uma característica comum nessa classe de critérios baseados em análise de fluxo de dados é a de que eles requerem que sejam testadas as interações que envolvam definições de variáveis de programas e subsequentes referências a estas definições [HER76, LAS83, RAP82, RAP85, NTA84, URA88]. Uma motivação para a introdução dos critérios baseados na análise de fluxo de dados foi a indicação de que, mesmo para programas pequenos, o teste de ramos não revela a presença de erros simples e triviais; e que em geral, o teste de todos os caminhos é impraticável. Esses critérios têm origem na intuição de que, assim como não se deve considerar suficientemente testado um programa que não tenha todos os seus comandos exercitados, não se deve considerar suficientemente testado um programa se todos os resultados computacionais não tiverem sido usados pelo menos uma vez [RAP85].

O objetivo principal da introdução dos critérios baseados em fluxo de dados é fornecer uma hierarquia de critérios entre os critérios todos os ramos e todos os caminhos e tornar o teste estrutural mais rigoroso. Do ponto de vista do teste de funções embutidas num programa, critérios que usam informações de fluxo de dados para derivar os requisitos de teste são mais adequados do que aqueles que utilizam apenas informações de fluxo de controle, uma vez que os primeiros identificam dependências de dados e portanto, requerem segmentos funcionais [HOW87, URA88].

A seguir, introduzem-se a terminologia e os conceitos básicos pertinentes ao teste estrutural de programas. Outros trabalhos de pesquisa relacionados com o tema principal da tese são discutidos na Seção 2.2.

2.1 Teste Estrutural — Terminologia e Conceitos Básicos

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 (grafo de programa) $G = (N, E, s)$ consiste em estabelecer uma correspondência entre nós e blocos e, em indicar possíveis fluxos de controle entre blocos através dos arcos. Considera-se todo grafo de programa como um grafo dirigido com um único nó de entrada $s \in N$ e um único nó de saída $e \in N$.

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

completo é um caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo G.

Seja $IN(x)$ e $OUT(x)$ o número de arcos que entram e que saem do nó x, respectivamente. Se $IN(x) = 0$, x é um *nó de entrada*, e se $OUT(x) = 0$, x é um nó de saída.

As ocorrências de uma variável em um programa podem ser uma *definição de variável*, um *uso de variável* ou uma *indefinição*. Usualmente, os tipos de ocorrências de variáveis são definidos por um *modelo de fluxo de dados*.

Conforme o modelo de fluxo de dados definido na Seção 4.3, uma *definição de variável* ocorre quando um valor é armazenado em uma posição de memória. Em geral, em um programa, uma ocorrência de variável é uma definição se ela está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; ou iii) em chamadas de procedimentos como parâmetro de saída. A passagem de valores entre procedimentos através da passagem de parâmetros pode ser por: *valor*, *referência* ou por *nome* [GHE87]. Se a variável for passada por referência ou por nome considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são distinguidas das demais e são ditas *definidas por referência*.

A ocorrência de uma variável é um *uso* quando a referência a essa variável não a estiver definindo. Dois tipos de usos são distinguidos – *c-uso* e *p-uso*. O primeiro tipo afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado. O segundo tipo afeta diretamente o fluxo de controle do programa.

Uma variável está *indefinida* quando, ou não se tem acesso ao seu valor, ou sua localização deixa de estar definida na memória.

Um caminho $(i, n_1, \ldots, n_m, j)$, $m \geq 0$ que não contenha definição de uma variável nos nós $n_1, \ldots, n_m$ é chamado de *caminho livre de definição* com respeito a (c.r.a) x do nó i ao nó j e do nó i ao arco (n_m, j).

Um nó i possui uma *definição global* de uma variável x se ocorre uma definição de x no nó i e existe um caminho livre de definição de i para algum nó ou para algum arco que contém um c-uso ou um p-uso, respectivamente, da variável x. Um c-uso da variável x em um nó j é um *c-uso global* se não existir uma definição de x no nó j precedendo este c-uso; caso contrário é um *c-uso local*.

Para estabelecer a Família de Critérios de Fluxo de Dados (DFCF), Rapps e Weyuker [RAP82, RAP85] introduziram alguns conceitos e definições; um deles foi a proposição do *grafo def-uso ('def-use graph')* que consiste em uma extensão do grafo de fluxo de controle, incorporando-se informações semânticas do programa a este grafo. O grafo def-uso é obtido a partir do grafo de programa associando-se a cada nó i os conjuntos *c-use*(i) = { variáveis com c-uso global no bloco i } e *def* (i) = { variáveis

com definições globais no bloco i }, e a cada arco (i,j) o conjunto *p-use* (i,j) = { variáveis com p-usos no arco (i,j)}. Dois conjuntos foram definidos: *dcu* (x,i) = { nós j tal que $x \in$ c-use (j) e existe um caminho livre de definição c.r.a. x do nó i para o nó j} e, *dpu* (x,i) = { arcos (j,k) tal que $x \in$ p-use (j,k) e existe um caminho livre de definição c.r.a. x do nó i para o arco (j,k)}. Adicionalmente, definiram: *du-caminho*, *associação definição-c-uso*, *associação definição-p-uso* e *associação*.

Um caminho $(n_1, n_2, \ldots, n_j, n_k)$ é um *du-caminho* c.r.a. variável x se n_1 tiver uma definição global de x e: (1) ou n_k tem um c-uso de x e $(n_1, n_2, \ldots, n_j, n_k)$ é um caminho simples livre de definição c.r.a. x; ou (2) (n_j, n_k) tem um p-uso de x e $(n_1, n_2, \ldots, n_j, n_k)$ é um caminho livre de definição c.r.a. x e $n_1, n_2, \ldots, n_j$ é um caminho livre de laço[1].

Uma *associação definição-c-uso* é uma tripla (i,j,x) onde i é um nó que contém uma definição global de x e $j \in dcu(x,i)$. Uma *associação definição-p-uso* é uma tripla $(i,(j,k),x)$ onde i é um nó que contém uma definição global de x e $(j,k) \in dpu(x,i)$. Uma *associação* é uma associação definição-c-uso, uma associação definição-p-uso ou um du-caminho.

A Família de Critérios de Fluxo de Dados, assim como os demais critérios baseados em fluxo de dados, para requererem um determinado elemento (caminho, associação, etc.) exigem a ocorrência explícita de um uso de variável; este aspecto é que determina as principais limitações dessa classe de critérios, como será visto posteriormente.

O conceito de *potencial uso*, introduzido inicialmente em [MAL88a, MAL88b], introduz pequenas mas fundamentais modificações nos conceitos apresentados acima. Os elementos a serem requeridos são caracterizados independentemente da ocorrência explícita de uma referência a uma determinada definição; se um uso dessa definição pode existir — *um potencial uso* — a 'potencial associação' entre a definição e o potencial uso é caracterizada, e eventualmente requerida. Na realidade, pode-se dizer que, com a introdução do conceito potencial uso, procura-se explorar todos os possíveis efeitos a partir de uma mudança de estado do programa em teste, decorrente de definição de variáveis em um determinado nó i. No Capítulo 3 ver-se-á que essas modificações influem fortemente nas propriedades e características desses critérios.

Definem-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 do nó i para o arco (j,k)}; *potencial-du-caminho* c.r.a variável x como um caminho livre de definição $(n_1, n_2, \ldots, n_j, n_k)$ c.r.a x do nó n_1 para o nó n_k e para o arco (n_j, n_k), onde o caminho $(n_1, n_2, \ldots, n_j)$ é um caminho livre de laço 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 $x \in defg(i)$ and $(j,k) \in pdpu(x,i)$;

[1]Rapps e Weyuker [WEY84] requerem no caso 2 que o caminho $(n_1, n_2, \ldots, n_j)$ seja um caminho livre de laço e livre de definição c.r.a. x. Isto implica que o caminho $(n_1, n_2, \ldots, n_j, n_k)$, com uma definição de x no nó n_j, seja considerado um du-caminho. A reformulação desta definição não visa a alterar o significado da definição original.

e *potencial associação* como uma associação potencial-definição-c-uso, uma associação potencial-definição-p-uso ou um potencial-du-caminho. Observe-se que *toda associação* é uma potencial-associação.

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

A cada nó i tal que $defg(i) \neq \phi$ é associado um grafo denominado $grafo(i)$; este grafo contém todos os potenciais-du-caminhos com início no nó i. Cada nó k do grafo(i) é completamente identificado pelo número do nó do grafo de programa que lhe deu origem e, pelo conjunto $deff(k)$ — o conjunto de variáveis definidas no nó i, mas que não são redefinidas num caminho do nó i para o nó k. Observe-se que $deff(k) \subseteq defg(i)$ e que $deff(i) = defg(i)$. Em [MAL88b] um algoritmo para obter esses grafos foi proposto; esses grafos e o conceito de arcos primitivos são a base para o Modelo de Descrição dos elementos requeridos, proposto no Capítulo 4. Na geração dos grafo(i), definições por referência são consideradas como uma definição no nó i, mas não são consideradas como redefinições em qualquer outro nó $j \neq i$, o que caracteriza um enfoque conservador.

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

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

Um caminho completo π cobre uma associação potencial-definição-c-uso $[i,j,x]$ [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 j [respectivamente, de 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 associação ou uma potencial-associação do nó i para o nó j [respectivamente, 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 j [respectivamente, para o arco (j,k)], que é incluído em Π.

Seja $\pi = (n_1,n_2,\ldots,n_k)$ um du-caminho (potencial-du-caminho) c.r.a.. x; define-se a *extensão a ciclo de* (π,x) como o conjunto de caminhos livre de definição c.r.a. x do seguinte tipo $(\lambda_1,\lambda_2,\ldots,\lambda_k)$ onde cada λ_i é um caminho de comprimento maior ou igual a 1 (um), com início e término no nó n_i. Deve-se observar que para qualquer du-caminho [potencial-du-caminho] π c.r.a. x, π pertence à *extensão a ciclo*(π,x).

Um *caminho completo é executável ou factível* se existe um conjunto de valores, que possa ser atribuído às variáveis de entrada do programa, que causa a execução desse caminho; caso contrário, diz-se que ele é *não executável* [FRA87]. Um *caminho é executável* se ele for um subcaminho de um caminho completo executável, isto é, se ele for 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: $fdcu(x,i) = \{j \in dcu(x,i) \mid$ *a associação* (i,j,x) *é executável*$\}$; $fdpu(x,i)=\{(j,k) \in dpu(x,i) \mid$ *a associação* $(i,(j,k),x)$ *é executável*$\}$; $fpdcu(x,i) = \{j \in pdcu(x,i) \mid$ *a potencial-associação* $[i,j,x]$ *é executável*$\}$; e, $fpdpu(x,i)=\{(j,k) \in pdpu(x,i) \mid$ *a potencial-associação* $[i,(j,k),x]$ *é executável*$\}$.

2.2 Trabalhos Relacionados

Conforme foi visto no início deste capítulo, os primeiros critérios de teste estrutural de programas usavam, essencialmente, informações de fluxo de controle para derivar os requisitos de teste; os mais conhecidos e utilizados são os critérios todos os ramos (teste de ramos) e o todos os nós (teste de comandos). A introdução dos critérios baseados em fluxo de dados tem como um dos objetivos, fornecer uma hierarquia de critérios entre os critérios todos os ramos e todos os caminhos pois, em geral, o teste de caminhos (aplicação do critério todos os caminhos) é impraticável.

O principal problema do teste de caminhos reside principalmente no fato de que, para muitos programas, o número de caminhos é infinito, ou extremamente grande, devido à ocorrência de estruturas de iteração no programa. Vários critérios têm sido propostos para a seleção de caminhos com o objetivo de introduzir critérios mais rigorosos que o critério todos os ramos e menos restritivo e dispendioso do que o critério todos os caminhos. Alguns, por exemplo, limitam-se a sugerir restrições no número de vezes que um laço é executado [KIN76]. Howden propôs o critério "boundary-interior" para classificar os caminhos em diferentes classes, dependendo da maneira como os laços são executados; ao menos um caminho que cubra cada classe é requerido [HOW75]. Woodward [WOO80], motivado pela indicação de estudos empíricos, de que a presença de um número significativo de erros pode ser revelada pela simples concatenação de arcos, propôs uma hierarquia de critérios que consiste essencialmente em concatenar seqüências de códigos terminadas por uma transferência de controle (LCAJ — "Linear Code Sequence And Jump"). Note-se que essas abordagens consideram ainda, essencialmente, a estrutura de controle para derivar os requisitos de teste. Howden observa que, mesmo que se limite o número de iterações de um laço a duas ou três vezes, ainda assim, frequentemente ter-se-á um número enorme de caminhos a serem testados. Neste sentido, os critérios baseados em análise de fluxo de dados são mais seletivos, além de promoverem a concatenação de arcos, e conduzem à seleção de caminhos com uma maior relação com os aspectos funcionais do programa [HOW87].

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 subsequentes referências a essas 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 subsequentes; todos eles requerem a ocorrência explícita de um uso de uma definição de variável para caracterizar e requerer essa interação.

Estudos comparativos entre os critérios baseados em fluxo de dados têm sido conduzidos apoiados principalmente por uma relação de inclusão e pelo estudo da complexidade dos critérios [CLA85, RAP85, WEY84,NAT88]. A *complexidade de um critério* C é definida como o número máximo de casos de teste requerido pelo critério no pior caso; ou seja, dado um programa qualquer P, se existir um conjunto de casos de teste T que seja $C-adequado$ para P, então existe um conjunto de casos de teste T_1, tal que a cardinalidade de T_1 é menor ou igual à complexidade do critério C. Os resultados desses estudos indicam que a maioria dos critérios de teste baseados em fluxo de dados tem complexidade de ordem exponencial e que alguns deles têm complexidade polinomial [WEY84]; no entanto, mostra-se no Capítulo 3, que todos eles têm complexidade de ordem exponencial.

A relação de inclusão estabelece uma ordem parcial entre os diversos critérios; portanto, como resultado da análise de inclusão obtém-se uma ordem parcial entre esses critérios. Diz-se que um critério de teste c_1 *inclui* um critério de teste c_2 se, para *qualquer* grafo de fluxo de controle G (qualquer programa P), *qualquer* conjunto de caminhos completos T que seja $c_1 - adequado$ para P seja $c_2 - adequado$ para P; ou seja, se T satisfaz c_1 também satisfaz c_2. O critério c_1 *inclui estritamente* um critério c_2, denotado por $c_1 \Rightarrow c_2$, se c_1 inclui c_2 e para algum grafo G existe um conjunto de caminhos completos T de G que satisfaz c_2 mas não satisfaz c_1. Se nem $c_1 \Rightarrow c_2$ nem $c_2 \Rightarrow c_1$, diz-se que esses critérios são *incomparáveis*. Denota-se por $c_1 \overset{A}{\Rightarrow} c_2$ quando a relação de $c_1 \Rightarrow c_2$ é válida somente para programas que satisfazem uma determinada propriedade A.

Herman pode ser considerado um dos precursores do uso de informações de fluxo de dados para o estabelecimento de critérios de teste [HER76]; na essência, o *critério de Herman* requer que toda referência (uso) a uma variável seja exercitada, pelo menos uma vez, a partir de pontos do programa em que essa variável fosse definida; ou seja, Herman estabelece associações entre definições e subsequentes usos (sem distinção do tipo de uso) de variáveis, e requer que essas associações sejam exercitadas pelos casos de teste. Os critérios *todos-usos* da FCDF de Rapps e Weyuker [RAP85], um dos critérios de Ntafos [NTA84] e um dos critérios de Laski e Korel [LAS83] são similares ao de Herman.

Introduzindo uma distinção entre tipos de referências a variáveis: uso computacional (c-uso) e uso predicativo (p-uso), Rapps e Weyuker introduziram uma Família de critérios de Fluxo de Dados (DFCF). Os três critérios centrais da DFCF são: *todas definições (all-defs)*, *todos os usos (all-uses)* e *todos-du-caminhos (all-du-paths)*. O critério todas definições requer que cada definição de variável seja exercitada pelo me-

nos por um p-uso ou um c-uso; o critério todos os usos requer que todas as associações entre uma definição e subsequentes c-usos ou p-usos dessa variável sejam exercitados pelos casos de teste; e o critério todos-du-caminhos requer que toda associação entre uma definição de variável e subsequentes p-usos ou c-usos dessa variável sejam exercitados por todos os caminhos livres de definição e livres de laço que cubram essa associação. Finalmente, os outros três critérios da DFCF — *todos-p-usos (all-p-uses)*, *todos-p-usos/alguns-c-usos (all-p-uses/some-c-uses)* e *todos c-usos/alguns-p-usos (all-c-uses/some-p-uses)* — são variações do critério todos os usos.

Ntafos introduz uma família de critérios denominada *K-tuplas requeridas (required K-tuples)*; essa família de critérios requer que todas as seqüências de (K-1), ou menos, interações definição uso (*2-dr interations*) sejam testadas. Variando-se K, obtém-se a família de critérios K-tuplas requeridas. Ntafos prova que esses critérios *incluem* o critério todos os usos e são incomparáveis com o critério todos-du-caminhos.

Laski e Korel introduzem outros critérios de teste que usam informações de fluxo de dados: o critério *ambiente de dados (data environment)*, o critério *contexto elementar de dados (elementary data context)* e o critério *contexto ordenado de dados (ordered data context)*. Um *contexto elementar de dados de um comando* s_i é definido por um subconjunto de variáveis $EDC(i)$ usadas no comando s_i, tal que existe um caminho do início do programa até s_i que inclua, para cada variável v pertencente a $EDC(i)$, caminhos livres de definição c.r.a v do ponto em que v é definida até s_i. A introdução destes critérios é baseada na intuição de que valores atribuídos a uma variável em um ponto particular i do programa pode depender de valores de várias variáveis, onde cada uma tenha várias definições distintas que atinjam s_i por caminhos livres de definição [LAS83]. Ntafos mostra que os critérios de Laski e Korel *não incluem* o critério todos os ramos, mesmo sem considerar os aspectos de não executabilidade de caminhos.

Mais recentemente, Ural e Yang introduziram um critério denominado *todos OI-caminhos simples (all-simple OI-paths)*. Este critério procura explorar efeitos entre entradas e saídas do programa. A idéia básica é estabelecer concatenações de associações entre definições e usos de variáveis que levem à determinação de caminhos completos; para isso Ural e Yang estabelecem seqüências (concatenação) de associações, definindo quando o uso de uma variável x é afetado pela definição de uma variável y. Ural e Yang provam que, para um *programa bem formado (well-formed-program)*, o critério todos OI-caminhos simples *inclui* o critério todos-du-caminhos da família de critérios de Rapps e Weyuker.

A análise de inclusão, conduzida por Frankl e Weyuker [FRA88], aplica-se à classe de programas P que satisfazem as propriedades NSUP — "No-Syntatic-Undefined-P-Use Property"— e NSL — "No-Straight-Line Property". Essas propriedades são requeridas em decorrência da própria definição da Família de Critérios de Fluxo de Dados (DFCF). Como resultado dessa análise, concluem que a DFCF estabelece uma *hierarquia de critérios* entre os critérios todos os ramos e todos os caminhos. No Capítulo 3, faz-se a comparação desses critérios com os critérios Potenciais Usos. Ntafos faz uma comparação mais abrangente, envolvendo alguns critérios baseados essencialmente em

fluxo de controle [NTA88].

Segundo Frankl e Weyuker [FRA86], uma das propriedades que deve ser satisfeita por um bom critério de teste é a *aplicabilidade*[WEY86, WEY88]; 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 inclui cada elemento exigido pelo critério C. Nenhum dos critérios de teste estrutural satisfaz essa propriedade [FRA87] pois algum caminho requerido pelo critério pode ser não executável. Note-se que é indecidível se existe um conjunto de casos de teste T que exercite todos os elementos requeridos por um dado critério de teste estrutural.

A presença de caminhos não executáveis modifica consideravelmente algumas das propriedades dos critérios; por exemplo, Frankl [FRA87, FRA88] observou que a relação de inclusão entre os critérios da Família de Critérios Fluxo de Dados muda significativamente: nenhum deles inclui o critério todos os ramos, por exemplo, não preenchendo algumas das propriedades esperadas de um "bom"critério de teste [WEY86, WEY88].

Na prática, ao aplicar-se um critério de teste estrutural aplica-se, na realidade, o correspondente critério estrutural executável. Portanto, é de fundamental importância a definição e o estudo das propriedades dos critérios na presença de caminhos não executáveis. Critérios baseados em fluxos de dados têm sido modificados, e estudados para lidar com este aspecto. Por exemplo, Frankl e Weyuker [FRA85, FRA87] introduzem uma família de critérios chamada "Feasible Data Flow Criteria". Frankl propõe algumas heurísticas para auxiliar na determinação de caminhos não executáveis [FRA87].

Considerando-se um exemplo modificado de Frankl [FRA87], ilustrado na Figura 3.2.1 do Capítulo 3, conclui-se que nenhum dos critérios baseados em fluxo de dados inclui o critério todos os ramos na presença de caminhos não executáveis. Diz-se que esses critérios não estabelecem uma "ponte" (*bridge the gap*) entre os critérios todos os ramos e todos os caminhos.

Estudos empíricos têm sido conduzidos com o objetivo principal de investigar o custo do uso de critérios de testes baseados em fluxo de dados nas atividades de teste (por exemplo, vide [WEY90, WEY88b, BIE89]). O conjunto de programas utilizado por Weyuker [WEY90] — extraído do livro de Kernighan & Plauger [KER81] — tem sido encarado como um benchmark [BEI90]. Um outro objetivo desses estudos foi determinar uma maneira de estimar o número de casos de testes necessários para satisfazer um dado critério para um programa P a ser testado. Weyuker obteve evidências de que os critérios baseados em fluxo de dados são utilizáveis do ponto de vista prático, pois o número de casos de teste requerido pode ser visto como linear com relação ao número de comandos de decisão do programa, até mesmo no pior caso empírico [WEY90]. No Capítulo 6 apresenta-se um estudo semelhante para os Critérios Potenciais Usos.

Considerando que a *Família de Critérios Potenciais Usos (PU)*, introduzida no Capítulo 3, é fortemente inspirada na Família de Critérios de Fluxo de Dados (DFCF),

de Rapps e Weyuker [RAP82, RAP85], e que pode ser vista como uma extensão desta, definem-se a seguir, alguns dos critérios da DFCF, importantes para o contexto desta tese.

Definição da Família de Critérios Baseados em Fluxo de Dados

Critério todas-definições - Π satisfaz o critério todas definições se para todo nó $i \in N \mid def(i) \neq \phi$ Π incluir, para cada variável $x \in def(i)$, alguma associação $(i,j,x) \mid j \in dcu(x,i)$ ou alguma associação $(i,(j,k),x) \mid (j,k) \in dpu(x,i)$. Isto é, toda definição global é exercitada.

Critério Todos-usos - Π satisfaz o critério todos-usos se Π incluir todas as associações $(i,j,x) \mid j \in dcu(x,i)$ e todas associações $(i,(j,k),x) \mid (j,k) \in dpu(x,i)$ para cada nó $i \in N$ e para cada $x \in def(i)$.

Critério Todos-du-caminhos - Π satisfaz o critério todos-du-caminhos se Π incluir, para cada $i \in N \mid def(i) \neq \phi$, todos os du-caminhos de i para j c.r.a cada variável $x \in def(i)$ para todas as associações $(i,j,x) \mid j \in dcu(x,i)$ e todos du-caminhos de i para (j,k) c.r.a cada variável $x \in def(i)$ para todas as associações $(i,(j,k),x) \mid (j,k) \in dpu(x,i)$.

Critério (todas-definições)* - Π satisfaz o critério (todas-definições)* se para todo nó $i \in N \mid def(i) \neq \phi$ Π incluir, para cada variável $x \in def(i) \mid fdcu(x,i) \cup fdpu(x,i) \neq \emptyset$, alguma associação $(i,j,x) \mid j \in fdcu(x,i)$ ou alguma associação $(i,(j,k),x) \mid (j,k) \in fdpu(x,i)$.

Critério (Todos-usos)* - Π satisfaz o critério (todos-usos)* se Π incluir todas as associações $(i,j,x) \mid j \in fdcu(x,i)$ e todas associações $(i,(j,k),x) \mid (j,k) \in fdpu(x,i)$ para cada nó $i \in N$ e para cada $x \in def(i)$.

Critério (Todos-du-caminhos)* - Π satisfaz o critério (todos-du-caminhos)* se Π incluir, para cada $i \in N \mid def(i) \neq \phi$, todos os du-caminhos executáveis de i para j c.r.a cada variável $x \in def(i)$ para todas as associações $(i,j,x) \mid j \in fdcu(x,i)$ e todos du-caminhos executáveis de i para (j,k) c.r.a cada variável $x \in def(i)$ para todas as associações $(i,(j,k),x) \mid (j,k) \in fdpu(x,i)$.

2.2.1 Aspectos de Automação das Atividades de Teste Estrutural de Programas

A aplicação dos critérios baseados em análise de fluxo de dados sem o apoio de uma ferramenta automatizada é limitada a programas muito simples [KOR85]. A importância de tais ferramentas tem sido discutida por vários autores [NTA88, FRA85, WEY84, PRI87, LAS83, WHI85] e algum esforço para implementar tal classe de ferramenta tem sido feito [FRA87, KOR85, WHI85]. A existência de caminhos não executáveis é

um aspecto bastante restritivo para a automatização desses critérios, assim como das atividades de teste de uma forma geral [HER76, WOO80, HOW87, FRA87, MALE90]; heurísticas têm sido introduzidas para lidar com este aspecto [FRA87].

Na medida em que os sistemas de software cresceram em tamanho e complexidade, o esforço requerido para testar esses sistemas tem crescido muito além das expectativas, implicando altos custos e produtos com baixa confiabilidade. Tem-se verificado que embora gaste-se, em geral, até 50% do orçamento para desenvolvimento do software em atividades de testes, um número significativo de defeitos permanece sem serem detectados nos softwares liberados; esses defeitos normalmente têm um impacto grande na operação normal do sistema.

Essa situação ensejou o desenvolvimento de ferramentas automáticas para auxiliar na produção de testes efetivos e na análise dos resultados dos testes. As ferramentas de teste estrutural de software fazem, em quase sua totalidade, análise de cobertura, segundo algum critério de teste selecionado, de um conjunto de casos de teste. Algumas delas oferecem algum suporte para a obtenção dos dados de entrada necessários para satisfazer um particular requisito de teste; em geral, esse suporte é baseado em execução simbólica do programa em teste. Apesar de não auxiliarem diretamente na determinação das entradas de um programa, necessárias para a execução de caminhos específicos, a maioria das ferramentas de teste apresenta ao usuário quais os requisitos de teste exigidos para que os critérios sejam satisfeitos. Assim, de certa maneira, orientam e auxiliam os usuários na elaboração dos casos de teste.

As ferramentas comerciais de teste estrutural de software (RXVP80 [DEU80], TCAT e TCAT-Path [REI90]) representam o *estado da prática.* Assim, como as ferramentas comerciais suportam a aplicação de critérios estruturais baseados somente no grafo de fluxo de controle, o que se tem na indústria, é a utilização dessa classe de critérios de teste.

Na década de 80, com o surgimento dos critérios baseados em análise de fluxo de dados, começaram a surgir as ferramentas que dão apoio à aplicação desses critérios. Até o momento não está disponível comercialmente nenhuma ferramenta que utiliza critérios baseados em fluxo de dados; as ferramentas disponíveis são protótipos de pesquisa desenvolvidos em universidades. Chaim fornece uma descrição mais detalhada de diversas ferramentas de apoio ao teste estrutural [CHA91d].

2.3 Considerações Finais

Através de resultados teóricos e empíricos, diversos pesquisadores têm procurado abstrair algumas propriedades associadas a critérios de teste, em particular, algumas associadas a critérios de teste estrutural [WEY84, WEY86, WEY88, STU89, CLA85, RAP85, NT88].

A *complexidade* e a *relação de inclusão* utilizadas nesses estudos refletem em geral as propriedades básicas que devem ser consideradas no processo de definição de um critério C, ou seja, o critério deve: i) incluir o critério *todos os ramos*; ii) requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; e iii) requerer um conjunto de casos de teste finito. Estes fatores influenciam a escolha entre esses diversos critérios de teste.

Os critérios efetivamente utilizados na prática são os critérios de fluxo de dados executáveis; são esses critérios que captam os aspectos semânticos do programa [FRA87]. Observou-se que a presença de caminhos não executáveis muda, em geral, as características e propriedades dos critérios. Por exemplo, nenhum dos critérios baseados em fluxo de dados *inclui* o critério todos os ramos; isto implica que, eventualmente, um erro que poderia ser identificado com o uso deste último critério, poderia não o ser por um conjunto de casos de teste T, que satisfizesse os critérios baseados em análise de fluxo de dados. Um dos motivos para a introdução dos critérios baseados na análise de fluxo de dados foi a indicação de que, mesmo para programas pequenos, o teste de ramos não revela a presença de erros computacionais simples e triviais; e em geral, o teste de todos os caminhos é impraticável; em contrapartida, são mais adequados para revelar a presença de erros de domínio. Portanto, a inclusão do critério todos os ramos é um requisito essencial. Frankl mostra que para classes de programas que satisfazem a propriedade NFUP, um dos critérios DFCF inclui o critério todos os ramos; é indecidível se um dado programa satisfaz essa propriedade. Uma forma de contornar esse aspecto, ainda segundo Frankl, seria requerer que os programas satisfizessem a propriedade NA ("No-Anomalies", o que seria restritivo, uma vez que muitos programas "corretos" não satisfazem essa propriedade [FRA86, FRA87].

Isto motivou o estudo e a introdução de novos conceitos para a definição de novos critérios objetivando, principalmente, eliminar essa deficiência; nessa busca, outros pontos importantes foram ressaltados, como será visto no Capítulo 3. Com a introdução do conceito *potencial uso* mostra-se, naquele capítulo, que esse objetivo é atingido com a introdução dos critérios *Potenciais Usos*. Consequentemente, nenhum dos critérios de teste baseados em fluxo de dados inclui os critérios Potenciais Usos.

Um outro ponto que deve ser observado cuidadosamente é o número de casos de teste requerido pelo critério de teste. Em geral, do ponto de vista teórico, o número de casos de teste requerido, pelos critérios baseados em análise de fluxo de dados, é restritivo para as suas utilizações na prática. Observou-se uma série de incorreções nos trabalhos relativos a este tópico, e dada a sua importância, este tópico é investigado na tese.

Esse fato motivou a condução de estudos empíricos para mostrar a viabilidade da aplicação desses critérios em ambientes de produção de software (por exemplo, vide [WEY90, WEY88b, BIE89]). Weyuker obteve evidências de que o número de casos de teste requerido para satisfazer os critérios da DFCF pode ser visto como linear em relação ao número de comandos de decisão [WEY90]. No Capítulo 6 apresenta-se um estudo semelhante para os Critérios Potenciais Usos; explora-se a hipótese de que

outras características do produto em teste, além do número de comandos de decisão, são também influentes na determinação dos modelos de estimativas. Por exemplo, seria de se esperar que o número de variáveis utilizadas no programa fosse significativo na estimativa do número de casos de teste necessários para satisfazer os critérios baseados em análise de fluxo de dados.

Observou-se ainda que o uso desses critérios é restrito a programas pequenos, sem o apoio de uma ferramenta. As ferramentas disponíveis, de apoio à aplicação dos critérios baseados em fluxo de dados, são protótipos de pesquisa em desenvolvimento. Esse fato motiva a proposição e desenvolvimento de uma ferramenta de apoio aos Critérios Potenciais Usos (propostos no Capítulo 3). A disponibilidade de uma ferramenta é um mecanismo de introdução e disseminação dos conceitos, princípios, técnicas e métodos nela embutidos.

Capítulo 3

Critérios Potenciais Usos: Definição e Análise de Propriedades

O estabelecimento de estratégias, métodos e critérios de teste visa a fornecer uma maneira sistemática para selecionar um subconjunto relativamente pequeno do domínio de entrada — um conjunto de casos de teste T — e ainda assim ser efetivo quanto à meta principal das atividades de testes: revelar a presença de defeitos no programa.

Os critérios baseados em análise de fluxo de dados utilizam a análise de fluxo de dados [HEC77] como fonte de informação para derivar os requisitos de teste e requerem que as interações que envolvem definições de variáveis de programa e subsequentes referências a essas definições sejam testadas; portanto, esses critérios baseiam-se, para a derivação de casos de teste, nas associações entre a definição de uma variável e os seus possíveis usos subsequentes.

Estudos comparativos entre os critérios baseados em fluxo de dados têm sido conduzidos suportados principalmente por uma relação de inclusão, que estabelece uma ordem parcial entre esses critérios, e pelo estudo da complexidade dos critérios [CLA86, RAP85, WEY84,NAT88]. A complexidade de um critério é definida como o número máximo de casos de teste requerido, no pior caso. A maioria dos critérios de teste baseados em fluxo de dados tem complexidade de ordem exponencial, como será visto a seguir.

A complexidade e a relação de inclusão refletem em geral as propriedades básicas que devem ser consideradas no processo de definição de um critério C, ou seja, o critério deve:

i) incluir o critério *todos os ramos*, ou seja, um conjunto de casos de teste que exercite os elementos requeridos pelo critério C deve exercitar todos os ramos do programa;

ii) requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; isto equivale ao critério C incluir o critério *todas-def*; e

iii) requerer um conjunto de casos de teste finito.

A principal desvantagem dos critérios baseados em análise de fluxo de dados é que na presença de caminhos não executáveis eles não garantem a inclusão do critério todos os ramos. Deve-se ressaltar que a maioria dos programas reais contém caminhos não executáveis; este fato foi observado no benchmark aplicado nos critérios Potenciais Usos, com o uso da ferramenta POKE-TOOL, descrita no Cap. 5.

Neste Capítulo, definem-se a *Família de Critérios Potenciais Usos* (PU) e a correspondente *Família de Critérios Potenciais Usos Executáveis* (FPU). Estes critérios são baseados no conceito *potencial uso* ; os critérios Potenciais Usos Básicos foram introduzidos em [MAL88a] e são denominados: critérios *todos-potenciais-du-caminhos*, *todos-potenciais-usos* e *todos-potenciais-usos/du*. Estes requerem associações independentemente da ocorrência explícita de uma referência a uma determinada definição; se um uso dessa definição pode existir — *um potencial uso* — a potencial associação é requerida. Dito de outra maneira, requerem basicamente que caminhos livres de definição, em relação a qualquer nó i que possua 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 é realizada; isto está de acordo com a filosofia discutida por Myers [MYE79]: um erro está claramente presente se um programa não faz o que supõe-se que ele faça, mas erros estão também presentes se um programa faz o que supõe-se que não faça; adicionalmente, podem auxiliar na identificação de falhas causadas por dependências de fluxo de dados ausentes [POD90], originadas por usos de variáveis ausentes, como ilustrado nos exemplos da Seção 3.1.

Os critérios Potenciais Usos satisfazem os três requisitos básicos citados acima, ou seja, estabelecem uma hierarquia de critérios entre os critérios todos os ramos e todos os caminhos, mesmo na presença de caminhos não executáveis; além disto, nenhum outro critério de teste baseado em fluxo de dados inclui os critérios Potenciais Usos. Outro ponto importante, é que apesar de terem complexidade de ordem exponencial, como será visto na Seção 3.2, o que poderia ser um limitante para a aplicação efetiva destes critérios, assim como dos demais critérios baseados em análise de fluxo de dados, na prática, requerem um número pequeno de casos de teste, o que pôde ser observado pela aplicação do benchmark, discutido no Capítulo 6.

Na Seção 3.1 a Família de Critérios Potenciais Usos e a correspondente Família de Critérios Potenciais Usos Executáveis são definidas; alguns critérios da Família de Critérios Fluxo de Dados (FCFD) de Rapps e Weyuker foram definidos no Capítulo 2, uma vez que os critérios Potenciais Usos podem ser vistos como uma extensão deles. A comparação dos critérios Potenciais Usos com os critérios FCFD e com os demais critérios baseados em fluxo de dados é discutida na Seção 3.2.

3.1 Definição da Família de Critérios Potenciais Usos

Seja $G = (N, A, s)$ um grafo de fluxo de controle e Π um conjunto de caminhos completos de G. Definem-se:

Critérios Potenciais Usos Básicos

Critério Todos-potenciais-usos - Π satisfaz o critério todos-potenciais-usos (all-potential-uses) 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 de i . Equivalentemente, em termos do conceito de potencial associação, Π deve incluir, para cada $i \in N \mid defg(i) \neq \phi$, 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 $i \in N$ e para cada $x \in defg(i)$.

Critério Todos-potenciais-usos/du - Π satisfaz o critério todos-potenciais-usos/du (all-potential-uses/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 nó e para todo arco possível de ser alcançado a partir de i. Equivalentemente, em termos do conceito de potencial associação, Π deve incluir, para cada $i \in N \mid defg(i) \neq \phi$, um potencial-du-caminho de i para j c.r.a x para todas as potenciais associações $[i, j, x] \mid j \in pdcu(x, i)$ e 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 (all-potential-du-path) se para todo nó $i \in N \mid defg(i) \neq \phi$, Π inclui todos os *potenciais du-caminhos* c.r.a todas as variáveis $x \in defg(i)$ a partir do nó i para todo nó $j \in pdcu(x, i)$ e para todo arco $(j, k) \in pdpu(x, i)$. Equivalentemente, em termos do conceito de potencial associação, Π deve incluir, para cada $i \in N \mid defg(i) \neq \phi$, todos os potenciais-du-caminhos de i para j c.r.a cada variável $x \in defg(i)$ para todas as potenciais associações $[i, j, x] \mid j \in pdcu(x, i)$ e todos potenciais-du-caminhos de i para (j, k) c.r.a cada variável $x \in defg(i)$ para todas as potenciais associações $[i, (j, k), x] \mid (j, k) \in pdpu(x, i)$. [1].

Para exemplificar a aplicação dos critérios Potenciais Usos Básicos considere o programa da Figura 3.1, cujo grafo de fluxo de controle está representado na Figura 3.2. Os elementos requeridos pelos critérios todos-potenciais-usos e todos-potenciais-usos/du

[1] Apesar do conceito de potencial associação englobar o de potencial-du-caminho, quando nos referenciarmos a este critério, diremos que ele exige *caminhos* enquanto que os outros dois exigem *associações*.

estão incluídos na Figura 3.3; os requeridos pelo critério todos-potenciais-du-caminhos estão representados na Figura 3.4. Observe que na Figura 3.2 são fornecidos conjuntos de caminhos completos que satisfazem os critérios Potenciais Usos; na prática, em relação aos critérios correspondentes da Família de Critérios de Fluxo de Dados, os mesmos conjuntos teriam que ser elaborados. Considere, no entanto, uma versão incorreta deste programa, ilustrada na Figura 3.5 cujo grafo de fluxo de controle é ilustrado na Figura 3.6; os mesmos elementos seriam requeridos pelos critérios Potenciais Usos, porém, o caminho $(8, 9, 5, 6, 8)$, não seria requerido por nenhum dos critérios de Rapps e Weyuker, conforme é mostrado pelos conjuntos de casos de teste que satisfazem esses critérios.

Uma das propriedades que deve ser satisfeita por um bom critério de teste é a *aplicabilidade* [FRA86]; 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 inclui cada elemento exigido pelo critério C.

Os critérios Potenciais Usos foram modificados com o intuito de satisfazer a propriedade de aplicabilidade, uma vez que, a exemplo dos demais critérios de teste estrutural, não satisfazem essa propriedade, pois, entre os elementos requeridos podem existir elementos não executáveis. Em geral, é indecidível se um dado caminho é ou pode ser executável; consequentemente, é indecidível se existe um conjunto de casos de teste T que seja $C - adequado$ para um dado programa P. A presença de caminhos não executáveis modifica consideravelmente algumas das propriedades dos critérios; por exemplo, Frankl [FRA87, FRA88] observou que a relação de inclusão entre os critérios da Família de Critérios Fluxo de Dados muda significativamente. Os critérios que satisfazem a propriedade de aplicabilidade serão referenciados como *critérios executáveis.*

Na prática, ao aplicar-se um critério de teste estrutural aplica-se, na realidade, o correspondente critério estrutural executável. Portanto, é de fundamental importância a definição e o estudo das propriedades dos critérios na presença de caminhos não executáveis.

A modificação introduzida nos critérios Potenciais Usos consiste, essencialmente, em selecionar as potenciais-associações requeridas utilizando-se os conjuntos $fpdcu(x, i)$ e $fpdpu(x, i)$ ao invés dos conjuntos $pdcu(x, i)$ e $pdpu(x, i)$, respectivamente; selecionam-se, portanto, apenas os elementos executáveis; por exemplo, o caminho $(1, 3, 4, 5, 10, 11, 12)$ do exemplo da Figura 3.1 não seria requerido. Os correspondentes critérios, definidos a seguir, são denominados Critérios Potenciais Usos Executáveis — todos-potenciais-usos executáveis, todos-potenciais-usos/du executáveis e todos-potenciais-du-caminhos executáveis. Então, para cada critério Potencial Uso C define-se um novo critério C^*. Observe-se que, para qualquer critério Potencial Uso C, $C \Rightarrow C^*$. Segundo Frankl [FRA87, FRA88], essa modificação muda o problema da indecidibilidade sobre a existência de um conjunto de casos de teste T que seja $C - adequado$ para um dado programa P para o problema de reconhecer (avaliar) se um dado conjunto de casos de teste T é $C - adequado$ para um dado programa P.

Critérios Potenciais Usos Executáveis

Critério (Todos-potenciais-usos)* - Π satisfaz o critério (todos-potenciais-usos)* se, para todo nó $i \in N \mid defg(i) \neq \phi$ e para toda variável $x \mid x \in defg(i)$, Π incluir todas as potenciais associações $[i,j,x] \mid j \in fpdcu(x,i)$ e todas as potenciais associações $[i,(j,k),x] \mid (j,k) \in fpdpu(x,i)$.

Critério (Todos-potenciais-usos/du)* - Π satisfaz o critério (todos-potenciais-usos/du)* se, para todo nó $i \in N \mid defg(i) \neq \phi$ e para toda variável $x \mid x \in defg(i)$, Π incluir um potencial-du-caminho executável para todas as potenciais associações $[i,j,x] \mid j \in fpdcu(x,i)$ e um potencial-du-caminho executável para todas as potenciais associações $[i,(j,k),x] \mid (j,k) \in fpdpu(x,i)$.

Critério (Todos-potenciais-du-caminhos)* - Π satisfaz o critério (todos-potenciais-du-caminhos)* se para todo nó $i \in G \mid defg(i) \neq \phi$, Π incluir todos os *potenciais-du-caminhos executáveis* c.r.a todas as variáveis $x \in defg(i)$ a partir do nó i para todo nó $j \in fpdcu(x,i)$ e para todo arco $(j,k) \in fpdpu(x,i)$. Equivalentemente, em termos do conceito de potencial associação, Π deve incluir todos os potenciais-du-caminhos executáveis de i para j c.r.a cada variável $x \in defg(i)$ para todas as potenciais associações executáveis $[i,j,x] \mid j \in fpdcu(x,i)$ e todos os potenciais-du-caminhos executáveis de i para (j,k) c.r.a cada variável $x \in defg(i)$ para todas as potenciais associações executáveis $[i,(j,k),x] \mid (j,k) \in fpdpu(x,i)$.

critério (todos-nos)* - Requer a execução de todos os nós executáveis.

critério (todos-ramos)* - Requer a execução de todos os ramos executáveis.

critério (todos-caminhos)* - Requer a execução de todos os caminhos executáveis.

O critério todos-potenciais-du-caminhos foi estendido utilizando-se o conceito de *extensão a ciclo* [FRA86] com o objetivo principal de obter-se uma hierarquia de critérios que incluísse os critérios todos-ramos e todas-definições, mesmo na presença de caminhos não executáveis; observe-se que o critério todos-potenciais-usos é na realidade o critério todos-potenciais-usos/du estendido a ciclo. Os critérios Potenciais Usos Básicos e os critérios Potenciais Usos Estendidos a Ciclo constituem a *Família de Critérios Potenciais Usos* e a correspondente *Família de Critérios Potenciais Usos Executáveis*, respectivamente.

Critérios Potenciais Usos Estendidos a Ciclo

Critério Todos-potenciais-usos/du Estendido a Ciclo - Π satisfaz o critério todos-potenciais-usos/du estendido a ciclo para um dado programa P, se somente se, Π incluir, para cada potencial-associação $[i,j,x] \mid j \in pdcu(x,i)$ e para cada

potencial-associação $[i,(j,k),x] \mid (j,k) \in pdpu(x,i)$, algum caminho $\pi_1 \in$ extensão-a-ciclo(π,x), onde π é um potencial-du-caminho c.r.a. x do nó i para o nó j ou para o arco (j,k), respectivamente.

Critério Todos-potenciais-du-caminhos Estendido a Ciclo - Π satisfaz o critério todos-potenciais-du-caminhos estendido a ciclo se e somente se, para todo nó $i \in G \mid defg(i) \neq \phi$, Π incluir para cada variável $x \in defg(i)$ e para cada potencial-du-caminho π c.r.a. x a partir de i, algum caminho $\pi_1 \in$ extensão-a-ciclo(π,x).

Critérios Potenciais Usos Executáveis Estendido a Ciclo

Critério (Todos-potenciais-usos/du Estendido a Ciclo)* - Π satisfaz o critério todos-potenciais-usos/du estendido a ciclo para um dado programa P, se e somente se, Π incluir, para cada potencial-associação executável $[i,j,x] \mid j \in fpdcu(x,i)$ e para cada potencial-associação executável $[i,(j,k),x] \mid (j,k) \in fpdpu(x,i)$, algum caminho executável $\pi_1 \in$ extensão-a-ciclo(π,x), onde π é um potencial-du-caminho c.r.a. x do nó i para o nó j ou para o arco (j,k), respectivamente.

Critério (Todos-potenciais-du-caminhos Estendido a Ciclo)* - Π satisfaz o critério todos-potenciais-du-caminhos estendido a ciclo se e somente se, para todo nó $i \in G \mid defg(i) \neq \phi$, Π incluir para cada variável $x \in defg(i)$ e para cada potencial-du-caminho π c.r.a. x a partir de i, algum caminho executável $\pi_1 \in$ extensão-a-ciclo(π,x).

As propriedades e características básicas dos Critérios Potenciais Usos são discutidas na seção que se segue.

3.2 Análise de Propriedades: Comparação com outros Critérios de Teste

Estudos de propriedades e características de critérios de teste estrutural têm sido conduzidos, conforme discutido no Capítulo 2. As principais propriedades requeridas por um bom critério de teste podem ser extraídas das análises de complexidade e de inclusão apresentadas nas seções seguintes.

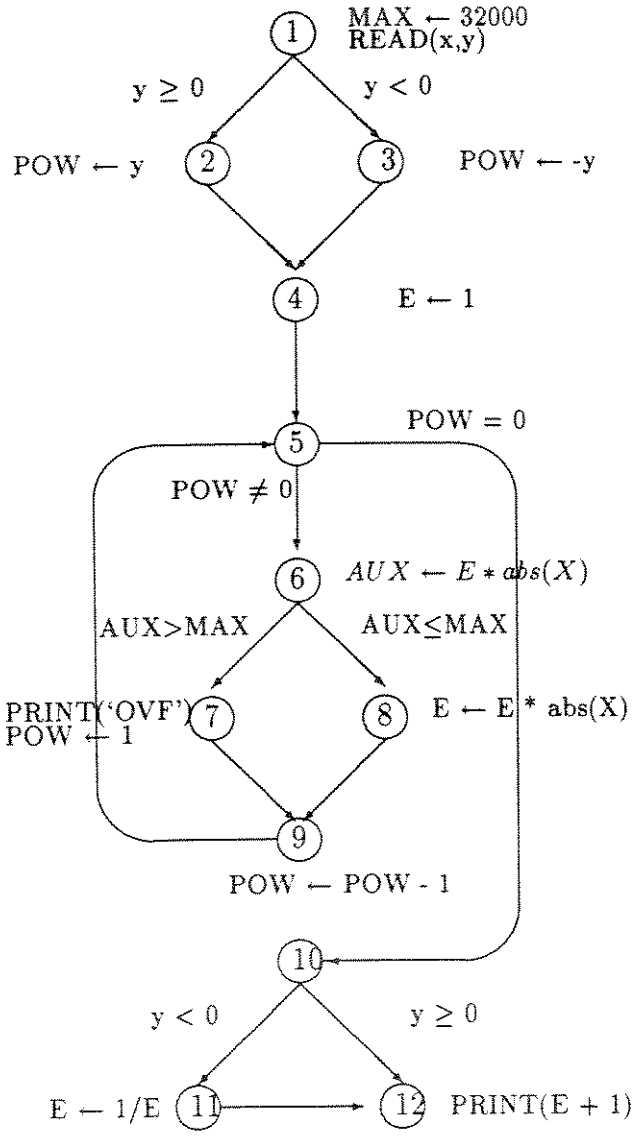
3.2.1 Análise de Inclusão

Nesta seção é apresentada uma síntese de comparação da Família de Critérios Potenciais Usos (Executáveis) com a Família de Critérios de Fluxo de Dados baseado na relação de inclusão discutida no Capítulo 2; essa comparação é detalhada em [MAL89b]. Adicionalmente, os critérios Potenciais Usos são comparados com os demais critérios de fluxo de dados. Como resultado da análise de inclusão obtém-se uma ordem parcial entre esses critérios. Um critério de teste c_1 *inclui* um critério de teste c_2 se, para *qualquer* grafo de fluxo de controle G (qualquer programa P), *qualquer* conjunto de caminhos completos T que seja $c_1 - adequado$ para P seja $c_2 - adequado$ para P; ou seja, se T satisfaz c_1 também satisfaz c_2. O critério c_1 *inclui estritamente* um critério c_2, denotado por $c_1 \Rightarrow c_2$, se c_1 inclui c_2 e para algum grafo G existe um conjunto de caminhos completos T de G que satisfaz c_2 mas não satisfaz c_1. Se nem $c_1 \Rightarrow c_2$ e nem $c_2 \Rightarrow c_1$, diz-se que esses critérios são *incomparáveis*. Denota-se por $c_1 \stackrel{A}{\Rightarrow} c_2$ quando a relação $c_1 \Rightarrow c_2$ é válida somente para programas que satisfazem uma determinada propriedade A.

A análise de inclusão, conduzida por Frankl e Weyuker [FRA88], aplica-se à classe de programas P que satisfazem as propriedades NSUP — "No-Syntatic-Undefined-P-Use Property"— e NSL — "No-Straigt-Line Property". Essas propriedades são requeridas em decorrência da própria definição da Família de Critérios de Fluxo de Dados (DFCF). Frankl e Weyuker observaram que, na presença de caminhos não executáveis, nenhum critério da família DFCF preenchia as propriedades esperadas de um 'bom' critério de teste; por exemplo, nenhum deles incluia o critério todos-ramos; restringindo-se a análise à classe de programas que satisfazem a propriedade "No-Feasible-Undefined-P-Uses Property" (NFUP) esse problema é contornado. No entanto, é indecidível se um dado programa satisfaz esta propriedade; uma alternativa seria restringir-se a classe de programas que satisfazem a propriedade 'No-Anomalies' (NA) o que, segundo Frankl e Weyuker, seria bastante restritivo uma vez que muitos bons programas não satisfazem necessariamente esta propriedade. Para os critérios Potenciais Usos, basta requerer-se, como será visto a seguir, a propriedade *'AT-Least-one-Definition-in-the-Entry-Node' (LDEN)*: definição de pelo menos uma variável de P no nó de entrada. Observe-se que a propriedade LDEN é facilmente preenchida pela maioria dos programas desenvolvidos. Em relação a essas propriedades, com exceção da NA, se um critério c_1 incluir estritamente um critério c_2 denota-se simplesmente por $c_1 \Rightarrow c_2$.

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

main()
{
 int x, y, pow;
 float aux, e;

 float max = 32000.0;
 scanf("%d %d", &x,&y);

 if(y>=0)
   pow = y;
 else /* y<0 */
   pow = -y;

e = 1;

while(pow != 0)
  {
   aux = e * abs(x);
   if(aux > max)
     {
      printf("Overflow error!\n");
      pow = 1;
     }
   else
     e = e * abs(x);
   pow = pow -1;
  }
  if(y<0)
     e = 1/e;
  printf("%d\n",e + 1);
 }
```

Figura 3.1: Exemplo de Aplicação dos Critérios Potenciais Usos



$\Pi_{all-pot-uses}$ =
{ (1,2,4,5,6,8,9,5,10,12), (1,3,4,5,6,7,9,5,10,11,12),
(1,2,4,5,10,12), (1,3,4,5,10,11,12),
(1,2,4,5,6,7,9,5,6,7,9,5,10,12),
(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)}

$\Pi_{all-pot-du-paths}$ =
$\Pi_{all-pot-uses}$ U
{ (1,2,4,5,10,11,12),(1,3,4,5,10,12),
(1,2,4,5,6,8,9,5,6,8,9,5,10,12) }

Figura 3.2: Grafo de Fluxo de Controle do Exemplo da Figura 3.1

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

1) <1,(1,3),{ x, y, max }>
2) <1,(10,12),{ x, y, max }>
3) <1,(10,11),{ x, y, max }>
4) <1,(6,8),{ x, y, max }>
5) <1,(9,5),{ x, y, max }>
6) <1,(6,7),{ x, y, max }>
7) <1,(1,2),{ x, y, max }>
8) <2,(10,12),{ pow }>
9) <2,(10,11),{ pow }>
10) <2,(8,9),{ pow }>
11) <2,(6,8),{ pow }>
12) <2,(6,7),{ pow }>
13) <3,(10,12),{ pow }>
14) <3,(10,11),{ pow }>
15) <3,(8,9),{ pow }>
16) <3,(6,8),{ pow }>
17) <3,(6,7),{ pow }>
18) <4,(10,12),{ e }>
19) <4,(10,11),{ e }>
20) <4,(6,8),{ e }>
21) <4,(9,5),{ e }>
22) <4,(6,7),{ e }>
23) <6,(6,8),{ aux }>
24) <6,(10,12),{ aux }>
25) <6,(10,11),{ aux }>
26) <6,(5,6),{ aux }>
27) <6,(6,7),{ aux }>
28) <7,(7,9),{ pow }>
29) <8,(10,12),{ e }>
30) <8,(10,11),{ e }>
31) <8,(6,8),{ e }>
32) <8,(7,9),{ e }>
33) <8,(6,7),{ e }>
34) <9,(10,12),{ pow }>
35) <9,(10,11),{ pow }>
36) <9,(8,9),{ pow }>
37) <9,(6,8),{ pow }>
38) <9,(6,7),{ pow }>
39) <11,(11,12),{ e }>
```

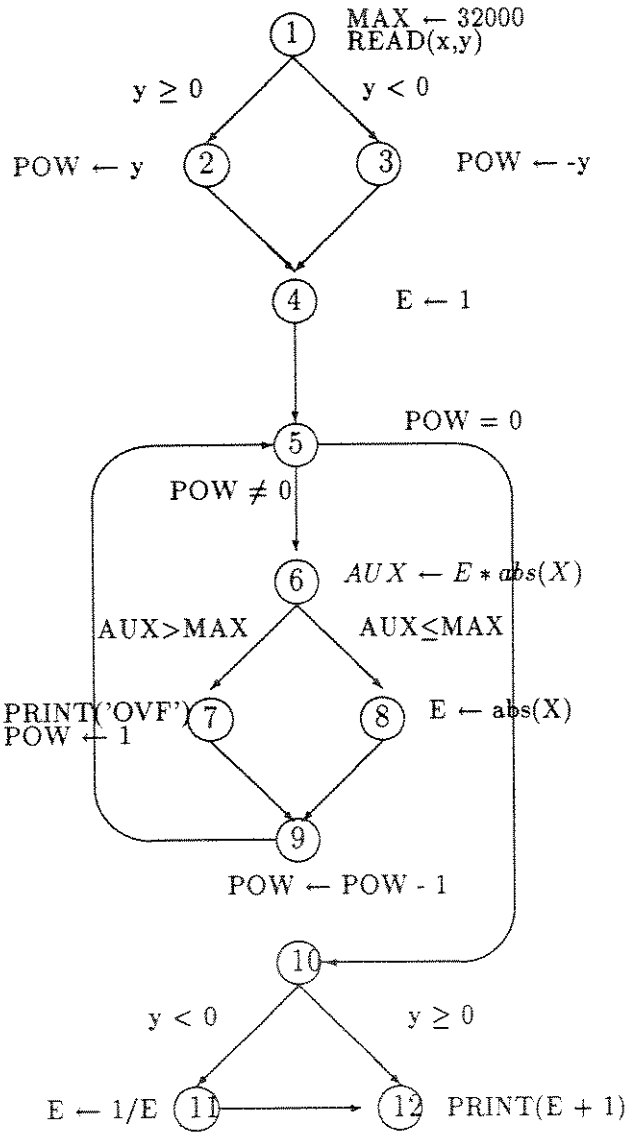
Figura 3.3: Elementos Requeridos pelos Critérios Todos-potenciais-usos e Todos-potenciais-usos/du

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS

 1) 1 3 4 5 10 12
 2) 1 3 4 5 10 11 12
 3) 1 3 4 5 6 8 9 5
 4) 1 3 4 5 6 7 9 5
 5) 1 2 4 5 10 12
 6) 1 2 4 5 10 11 12
 7) 1 2 4 5 6 8 9 5
 8) 1 2 4 5 6 7 9 5
 9) 2 4 5 10 12
10) 2 4 5 10 11 12
11) 2 4 5 6 8 9
12) 2 4 5 6 7
13) 3 4 5 10 12
14) 3 4 5 10 11 12
15) 3 4 5 6 8 9
16) 3 4 5 6 7
17) 4 5 10 12
18) 4 5 10 11
19) 4 5 6 8
20) 4 5 6 7 9 5
21) 6 8 9 5 10 12
22) 6 8 9 5 10 11 12
23) 6 8 9 5 6
24) 6 7 9 5 10 12
25) 6 7 9 5 10 11 12
26) 6 7 9 5 6
27) 7 9
28) 8 9 5 10 12
29) 8 9 5 10 11
30) 8 9 5 6 8
31) 8 9 5 6 7 9
32) 9 5 10 12
33) 9 5 10 11 12
34) 9 5 6 8 9
35) 9 5 6 7
36) 11 12
```

Figura 3.4: Elementos Requeridos pelos Critérios Todos-potenciais-du-caminhos

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

main()
{
 int x, y, pow;
 float aux, e;

 float max = 32000.0;
 scanf("%d %d", &x,&y);

 if(y>=0)
   pow = y;
 else /* y<0 */
   pow = -y;

e = 1;

while(pow != 0)
  {
   aux = e * abs(x);
   if(aux > max)
     {
      printf("Overflow error!\n");
      pow = 1;
     }
   else
     e = abs(x);
   pow = pow -1;
  }
  if(y<0)
     e = 1/e;
  printf("%d\n",e + 1);
 }
```

Figura 3.5: Versão Incorreta do Exemplo de Aplicação dos Critérios Potenciais Usos da Figura 3.1



$\Pi_{all-uses}$ = { (1,2,4,5,6,8,9,5,10,12), (1,3,4,5,6,7,9,5,10,11
(1,2,4,5,10,12), (1,3,4,5,10,11,12),
(1,2,4,5,6,7,9,5,6,7,9,5,10,12),
(1,2,4,5,6,8,9,5,6,7,9,5,10,12) }

$\Pi_{all-du-paths}$ =
$\Pi_{all-uses}$ U {(1,2,4,5,10,11,12),(1,3,4,5,10,12) }

$\Pi_{all-pot-uses}$ =
$\Pi_{all-uses}$ U { (1,2,4,5,6,8,9,5,6,8,9,5,10,12)}

$\Pi_{all-pot-du-paths}$ =
$\Pi_{all-du-paths}$ U { (1,2,4,5,6,8,9,5,6,8,9,5,10,12) }

CORRECT
E ←
$E * abs(X)$

Figura 3.6: Grafo de Fluxo de Controle do Exemplo da Figura 3.5

Os seguintes lemas são importantes para a compreensão dos resultados obtidos.

Lema 1 *Para qualquer programa P, se P não tiver caminhos não executáveis, então um conjunto de casos de teste T é C – adequado para P se e somente se T é C^* – adequado para P.*

Lema 2 *Se existe um caminho livre de definição π c.r.a. x de um nó i, que contém uma definição de x, para um nó j ou arco (j,k) contendo um uso de x, então existe um (não necessariamente único) du-caminho $\pi_1 \mid \pi \in$ extensão a ciclo (π_1, x).*

Lema 3 *Se existe um caminho livre de definição c.r.a. x do nó i para o nó j ou para o arco (j,k) então existe um potencial-du-path c.r.a. x do nó i para o nó j ou para o arco (j,k).*

Lema 4 *Para qualquer programa P, se P não tiver laços então um conjunto de casos de teste T é C – adequado para P se e somente se T é cyex – C – adequado para P.*

Teorema 1 *A família de critérios DFCF e a família de critérios Potenciais Usos (FPU) são parcialmente ordenadas pela relação de inclusão estrita conforme ilustrado na Figura 3.13. As famílias de critérios executáveis correspondentes, FDFCF e FPU, respectivamente, são parcialmente ordenadas conforme ilustrado na Figura 3.14. Adicionalmente, um critério c_i inclui um critério c_j se e somente se este fato é explicitamente caracterizado nas Figuras 3.13 e 3.14 ou este fato pode ser inferido a partir destas figuras considerando-se a propriedade de transitividade da relação de inclusão.*

Prova: Seja T um conjunto de caso de teste para um programa P (G o correpondente grafo de fluxo de controle), e seja Π o conjunto de caminhos executados por T. Em geral, para provar-se que a inclusão é estrita, contra-exemplos devem ser elaborados; os exemplos contidos nas Figuras 3.7, 3.8, 3.9, 3.10, 3.11 e 3.12 foram utilizados para este fim. Somente os aspectos mais relevantes da prova são apresentados (ver [MAL89b] para a prova completa).

- $(todos - potenciais - usos/du)^* \Rightarrow (todos - ramos)^*$

Suponha que T é $(todos - potenciais - usos/du)^* - adequado$ para P. Seja (i,j) um arco qualquer executável de P. Uma vez que o arco (i,j) é executável, existe pelo menos um caminho completo executável $\pi = (I, n_1, n_2, \ldots, i, j, \ldots F)$ do nó de entrada I para o nó de saída F tal que o arco (i,j) é incluído em π. A partir desta consideração resta mostrar, para completar a prova, que existe pelo menos um potencial-du-caminho executável a partir de um nó n_d para o arco (i,j) c.r.a. alguma variável v definida em n_d. Um desses potenciais du-caminhos executáveis será incluído em Π pois, T é $(todos - potenciais - usos/du)^* - adequado$: desta forma o arco (i,j) será incluído em Π.

i) Se o nó i tiver uma definição da variável v, então o arco (i,j) é um potencial-du-caminho executável do nó i para o arco (i,j).

ii) Considere-se que o caminho $\pi_i = (I, n_1, n_2, \ldots, n_k, i)$ seja um caminho livre de laço executável. Uma vez que o programa P satifaz a propriedade LDEN, o nó I possui pelo menos uma definição de alguma variável v. Se os nós $n_1, n_2, \ldots, n_k$ não tiverem redefinições da variável v definida no nó I, então o caminho $(I, n_1, n_2, \ldots, n_k, i, j)$ é um potencial-du-caminho executável c.r.a. v do nó I para o arco (i,j), por definição. Se algum nó n_d, $1 \leq d \leq k$, tiver uma redefinição de v, então o caminho $(n_d, n_{d+1}, \ldots, n_k, i, j)$ é um potencial-du-caminho executável c.r.a. v do nó n_d ao arco (i,j).

iii) Considere-se que o caminho $\pi_i = (I, n_1, n_2, \ldots, n_k, i)$ não seja um caminho livre de laços. Seja $(n_l, n_{l+1}, \ldots, n_{l+n}, n_l)$ o último laço no caminho π_i antes da ocorrência do nó i, i.e., $\pi_i = (I, n_1, n_2, \ldots, n_l, n_{l+1}, \ldots, n_{l+n}, n_l, n_s, \ldots, n_{s+m}, n_k, i)$. O caminho $\pi_{i1} = (n_l, n_s, \ldots, n_{s+m}, n_k, i)$ é um caminho livre de laços executável. Se em algum nó n_d do caminho π_{i1} ocorrer a definição de uma variável v, é imediata a conclusão de que o caminho $\pi_{ij1} = (n_l, n_s, \ldots, n_{s+m}, n_k, i, j)$ inclui um potencial-du-caminho executável c.r.a. v do nó n_d para o arco (i,j) (parte ii). Ainda, é fácil concluir que o conjunto de caminhos $\Pi_i = \{(n_{l+q}, \ldots, n_l, n_s, \ldots, n_k, i) \mid 1 \leq q \leq n\}$ contém somente caminhos livres de laços executáveis. Adicionalmente, um dos nós $n_l, n_{l+1}, \ldots, n_{l+n}$ deve possuir uma definição para alterar a condição do laço. Considere-se que tal definição ocorra no nó $n_d \in \{n_{l+q} \mid 1 \leq q \leq n\}$. Uma vez que o $(n_d, \ldots, n_l, n_s, \ldots, n_k, i) \in \Pi_i$, conclui-se que este caminho inclui um potencial-du-caminho executável do nó n_d para o arco (i,j) (parte ii). Se tal definição (que altera a condição do laço) ocorrer no nó n_l, tem-se simplesmente o caso de um caminho $\pi_{i1} = (n_l, n_s, \ldots, n_k, i)$ com uma definição no nó n_l.

• $(todos - potenciais - usos)^* \Rightarrow (todos - ramos)^*$

Suponha que T é $(todos - potenciais - usos)^* - adequado$ para P. Seja (i,j) um arco qualquer executável de P. Uma vez que o arco (i,j) é executável, existe pelo menos um caminho completo executável $\pi_c = (I, n_1, n_2, \ldots, n_k, i, j, \ldots, F)$ do nó de entrada nó I para o nó de saída F tal que o arco (i,j) está incluído em π_c. A partir desta consideração resta mostrar, para completar a prova, que existe pelo menos um caminho livre de definição executável de algum nó n_d para o arco (i,j) c.r.a. alguma variável v definida em n_d. Consequentemente, o conjunto Π deveria cobrir a associação $[n_d, (i,j), v]$, i.e., Π deveria incluir pelo menos um caminho livre de definição executável de n_d para o arco (i,j) c.r.a. v; desta forma Π incluiria o arco (i,j).

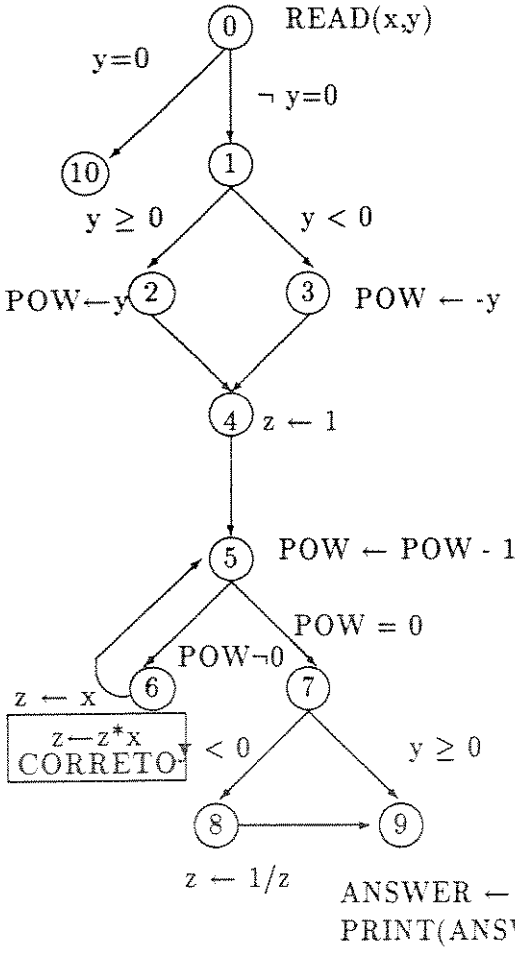
i) Se o nó i tiver a definição de uma variável v então o arco (i,j) é um caminho livre de definição executável do nó i para o arco (i,j) c.r.a. v.

ii) Uma vez que o programa P satisfaz a propriedade LDEN, o nó I possui pelo menos a definição de alguma variável v. Se os nós $n_1, n_2, \ldots, n_k$ não tiverem redefinições

da variável v, definida no nó I, tem-se que o caminho $(I, n_1, n_2, \ldots, n_k, i, j)$ é um caminho livre de definição executável do nó I para o arco (i, j), por definição. Se algum nó n_d, $1 \leq d \leq k$, possuir redefinição de v, então o caminho $(n_d, n_{d+1}, \ldots, n_k, i, j)$ é um caminho livre de definição executável c.r.a. v do nó n_d para o arco (i, j) .

Pode-se extrair da Figura 3.14 que alguns dos critérios Potenciais Usos, para a classe de programas que satisfazem a propriedade LDEN "bridge the gap" entre os critérios $(todos-ramos)^*$ e $(todos-caminhos)^*$ e ainda, preenchem o requisito básico, do ponto de vista de fluxo de dados, uma vez que todos eles incluem o critério $(all-defs)^*$. Por outro lado, nenhum entre os critérios DFCF "bridge the gap" entre os critérios $(todos-ramos)^*$ e $(todos-caminhos)^*$. Observe-se que fica assim estabelecida uma hierarquia de critérios entre os critérios $(todos-caminhos)^*$ e $(todos-ramos)^*$.

Em relação aos demais critérios de fluxo de dados, considere-se o exemplo da Figura 3.12, que consiste em um exemplo modificado de Frankl [FRA87]; assumindo, a exemplo de Frankl, uma análise de pior caso, ou seja, que os arcos $(5,6)$ e $(5,7)$ são ambos executáveis (considere-se, por exemplo, um ambiente no qual variáveis não iniciadas recebem valores arbitrários), conclui-se que os critérios de Herman, de Ntafos, de Laski e de Ural, na presença de caminhos não executáveis, não 'bridge the gap' entre os critérios $(todos-ramos)^*$ e $(todos-caminhos)^*$. Por exemplo, *o critério todos caminhos O/I simples* não requer o caminho $(1,2,4,5,7,8,9)$; observe-se que este é um caminho executável. Considerando-se as Figuras 3.6 e 3.12 e as definições dos critérios, a conclusão que os critérios Potenciais Usos e estes critérios são incomparáveis é imediata. Conclui-se, portanto, que nenhum critério de teste estrutural baseado em fluxo de dados, citados nesta tese, inclui os critérios Potenciais Usos.



Πall-nodes = {(0,10),(0,1,3,4,5,7,8,9),(0,1,2,4,5,6,5,7,8,9)}

Πall-edges = {(0,10),(0,1,3,4,5,6,5,7,8,9)}
Πall-defs = {(0,1,3,4,5,7,8,9),(0,1,2,4,5,6,5,7,8,9)}

Πall-uses = {(0,1,2,4,5,6,5,7,8,9),(0,10),
(0,1,3,4,5,6,5,7,9),(0,1,3,4,5,7,9)
(0,1,2,4,5,7,8,9)}

Π all-potential-uses = {(0,1,2,4,5,6,5,6,5,7,8,9),(0,10)
(0,1,3,4,5,6,5,7,9),(0,1,3,4,5,7,9)
(0,1,3,4,5,7,8,9)}

Πall-potential-uses/du = Πall-uses U
{(0,1,2,4,5,6,5,6,5,7,8,9)}

Πall-du-paths = Πall uses U {(0,1,3,4,5,7,8,9)
(0,1,2,4,5,7,9)}

Πall-potential-du-paths = Πall-du-paths U
{(0,1,2,4,5,6,5,6,5,7,8,9)}

Figura 3.7: Exemplo 1 para Análise de Inclusão

3.2.2 Análise de Complexidade

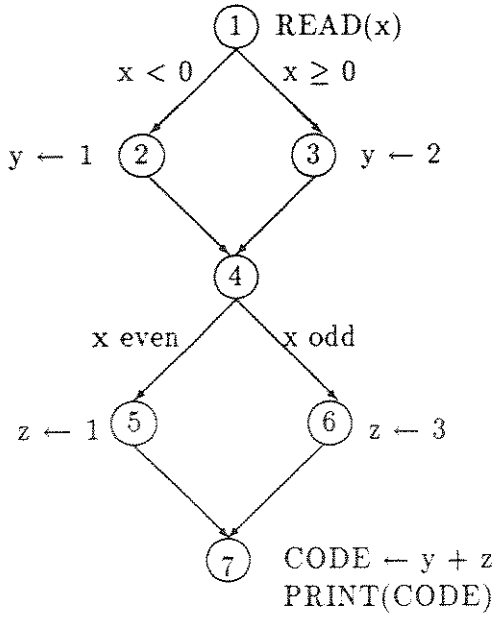
A *complexidade de um critério C* é definida como o número máximo de casos de teste requerido pelo critério no pior caso; ou seja, dado um programa qualquer P, se existir um conjunto de casos de teste T que seja $C-adequado$ para P, então existe um conjunto de casos de teste T_1 tal que a cardinalidade de T_1 é menor ou igual à complexidade do critério C. Observe-se que para a determinação da complexidade de um critério deve-se considerar, portanto, qualquer grafo de fluxo de controle G e qualquer distribuição de ocorrências de variáveis no programa, ou seja, para um fluxo de dados qualquer (para um grafo def qualquer).

Um primeiro passo para determinar a complexidade de um critério consiste em determinar-se uma estrutura de fluxo de controle, considerando-se um fluxo de dados qualquer, que maximize o número de elementos requeridos; no caso dos critérios Potenciais Usos, os elementos são potenciais-du-caminhos e associações. No Apêndice ?? mostra-se que o grafo de fluxo de controle da Figura 3.16 maximiza o número de potenciais-du-caminhos requeridos, dado por $((11/2)t+9)2^t-10t-9$ e requer 2^t casos de teste admitindo-se que o caminho $p_1 = (n_1, n_2, \ldots, n_i, \ldots, n_k, \ldots, n_i, \ldots, n_k, \ldots, n_i, \ldots, n_m)$ seja considerado como a concatenação dos potenciais-du-caminhos $(n_1, n_2, \ldots, n_i, \ldots, n_k)$, $(n_k, \ldots, n_i, \ldots, n_k)$ e $(n_k, \ldots, n_i, \ldots, n_m)$. Neste caso, admitindo-se que o caminho p_1 inclua estes potenciais-du-caminhos está se admitindo que a definição de uma variável x no nó n_k ocorra duas ou mais vezes antes de seu potencial uso no nó n_m. Um ponto importante a ressaltar é a facilidade de se elaborar programas exemplos com estruturas de controle consistindo de seqüências de comandos IF-THEN-ELSE que requeiram 2^t casos de teste. Consequentemente, a complexidade do critério todos-potenciais-du-caminhos é 2^t.

Com relação aos critérios todos-potenciais-usos e todos-potenciais-usos/du, sabe-se da análise de inclusão, que o critério todos-potenciais-du-caminhos inclui estes critérios; portanto, assume-se que 2^t é um limitante superior para a complexidade destes. Considerando o exemplo da Figura 3.17, verifica-se facilmente que 2^t casos de teste são requeridos e, consequentemente, a complexidade dos critérios todos-potenciais-usos e todos-potenciais-usos/du é 2^t. Observe-se que para a elaboração do exemplo da Figura 3.17 utilizou-se a idéia de sensibilização de caminhos utilizada na área de eletrônica, mais especificamente na área de circuitos digitais.

Os critérios estendidos a ciclo requerem no máximo 2^t casos de teste uma vez que esses critérios requerem somente um único caminho executável $\pi_1 \in$ extensão-a-ciclo(π, x) para cada potencial-du-caminho π do grafo de fluxo de controle.

Por outro lado, parece ser óbvio que os critérios Potenciais Usos Executáveis demandariam, no pior caso, um mesmo número de casos de teste que os correspondentes critérios Potenciais Usos. No entanto, a não executabilidade pode constituir uma restrição na combinação de potenciais-du-caminhos requeridos. Considerem-se os exemplos das Figuras 3.18 e 3.19; $2^t + 2^{t-1}$ casos de teste seriam requeridos para satisfazer os critérios todos-potenciais-du-caminhos. Utilizando-se a idéia de sensibilização de ca-



Πall-uses = {(1,2,4,5,7),(1,3,4,6,7)}

Πall-potential-uses = Πall-uses U {(1,2,4,6,7),(1,3,4,5,7)}

Πall-potential-uses/du = Πall potential-uses

Πall-du-paths = Πall-potential-du-paths

Figura 3.8: Exemplo 2 para Análise de Inclusão

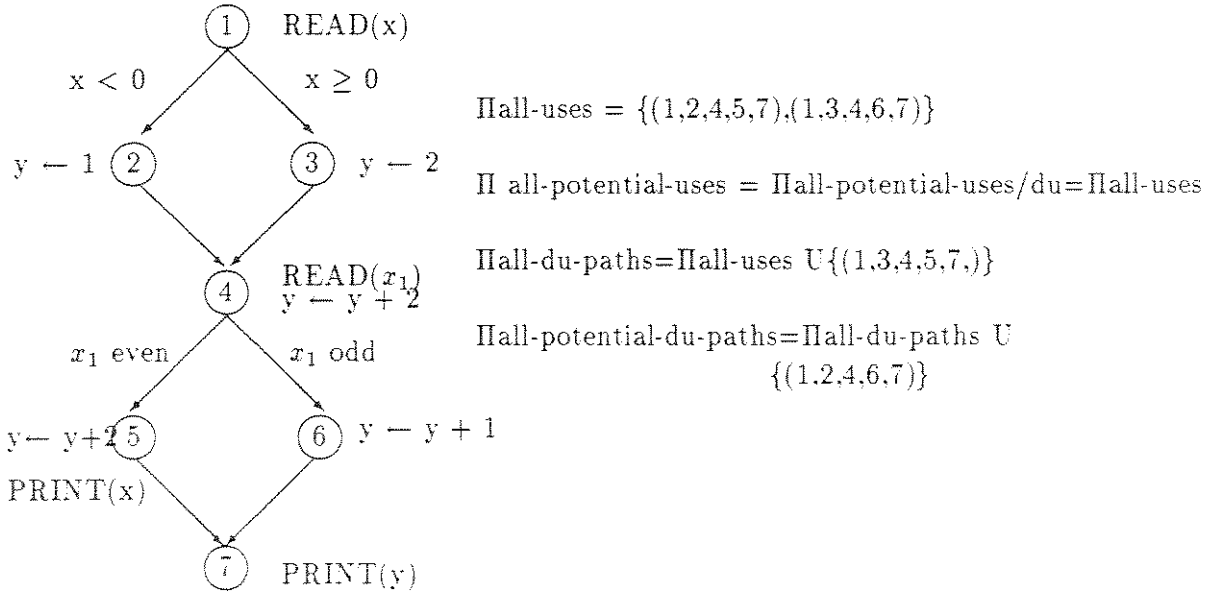
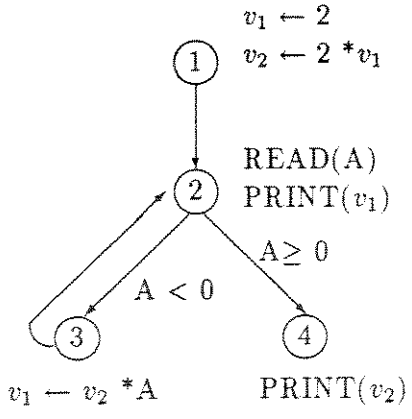


Figura 3.9: Exemplo 3 para Análise de Inclusão



Πall-uses = {(1,2,3,2,3,2,4)}

Πcycle-extended-potential-uses/du = {1,2,3,2,3,2,4)}

Πcycle-extended-potential-du-paths={(1,2,3.2,3,2,4)}

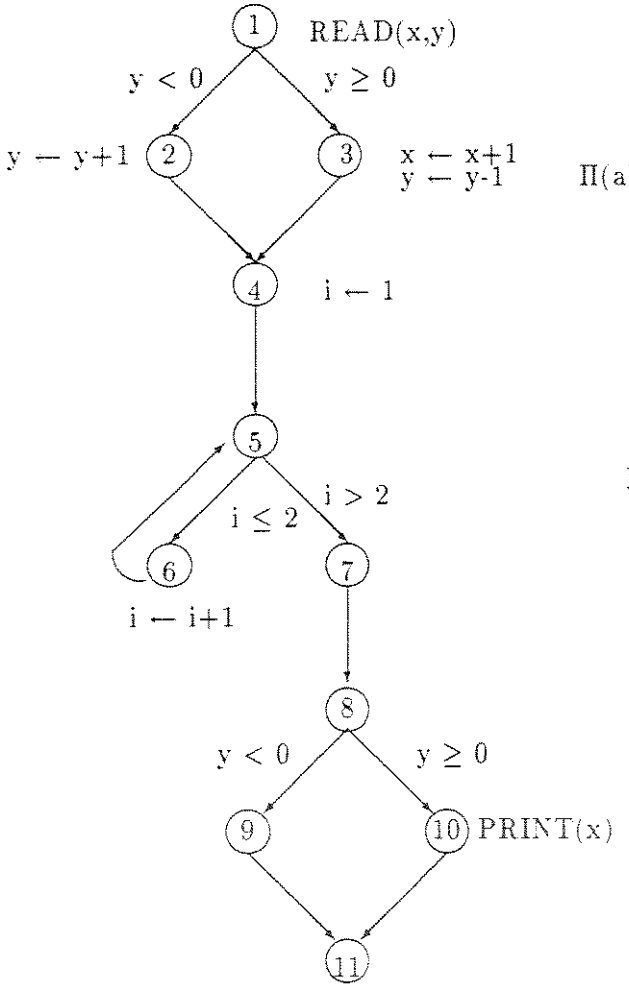
Πall-potential-du-paths={(1,2,3,2,3,2,4),(1,2.4)}

Πall-du-paths={(1,2,3,2,4),(1,2,4)}

Πcycle-extended-du-paths={(1,2,3,2,4)}

Πall-potential-uses/du={(1,2,3,2,3,2,4),(1,2.4)}

Figura 3.10: Exemplo 4 para Análise de Inclusão



Π(all-potential-du-paths)* = Π(all-potential uses/du)*=

{(1,2,4,5,6,5,6,5,7,8,10,11)

(1,3,4,5,6,5,6,5,7,8,9,11)}

Π (all-potential-uses)= Π (all-potential-du-paths)*U

{ (1,3,4,5,6,5,6,5,7,8,10,11)

(1,2,4,5,6,5,6,5,7,8,9,11)}

Figura 3.11: Exemplo 5 para Análise de Inclusão

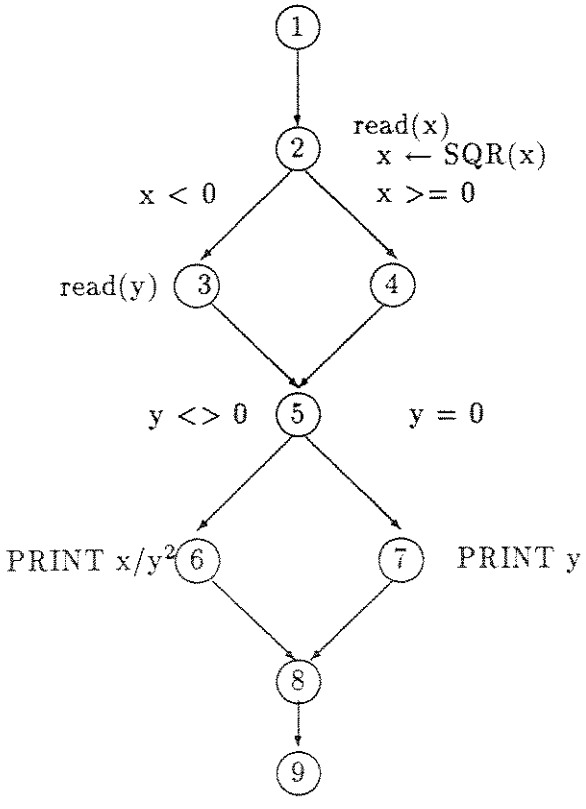


Figura 3.12: Exemplo para Análise de Inclusão na Presença de Caminhos não Executáveis

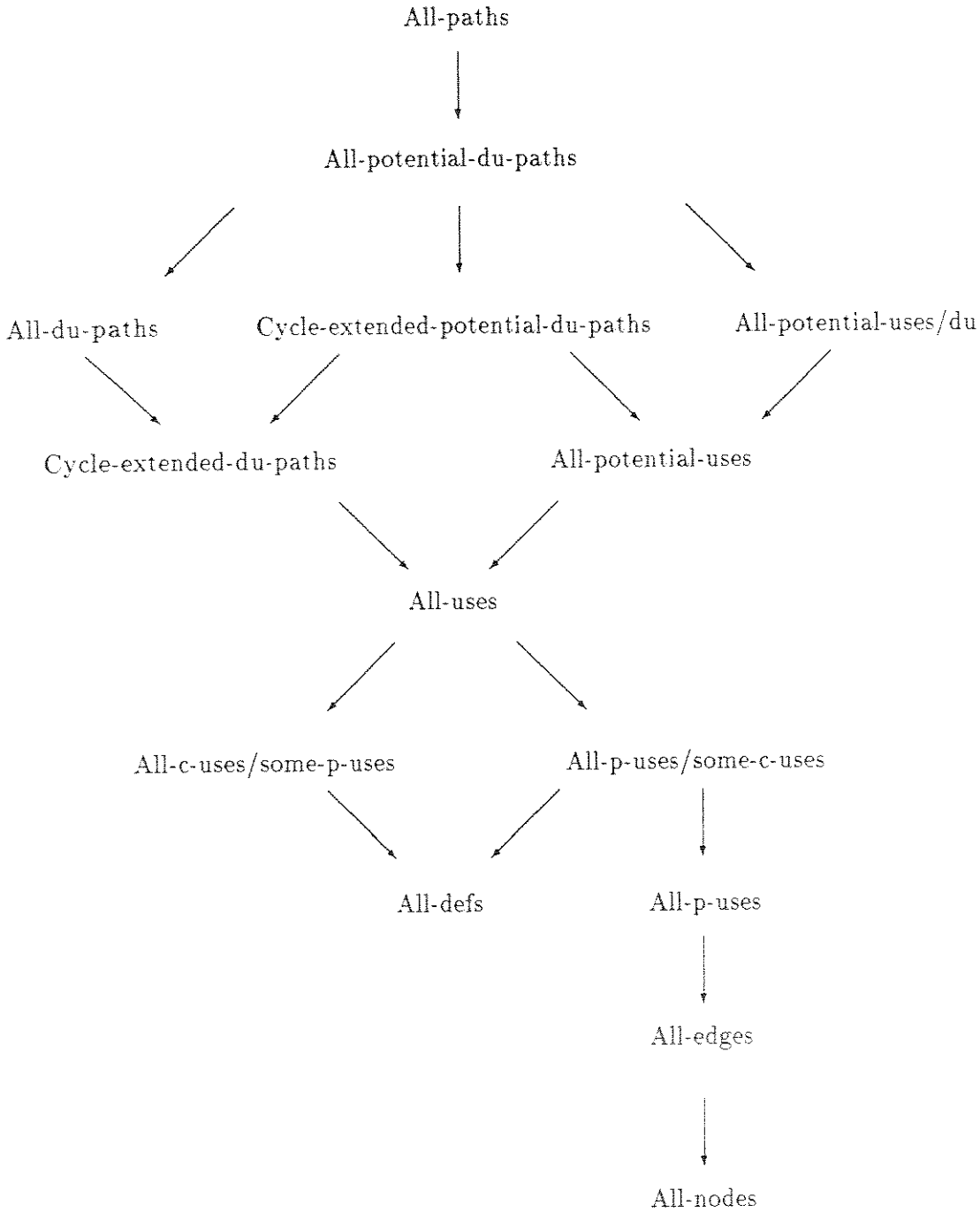


Figura 3.13: Ordem Parcial entre a Família de Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados (DFCF)

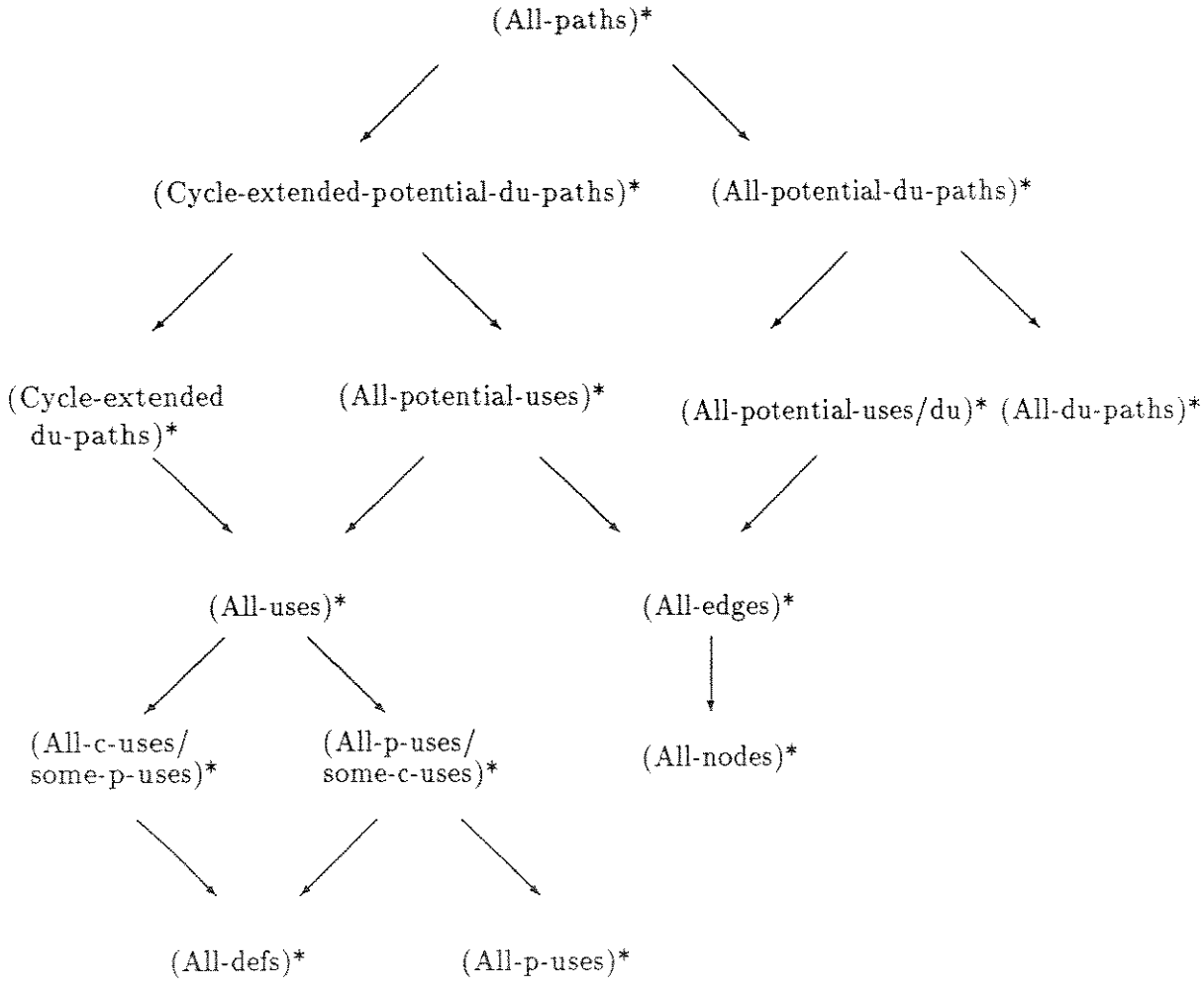


Figura 3.14: Ordem Parcial entre a Família de Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados (DFCF) na Presença de Caminhos não Executáveis

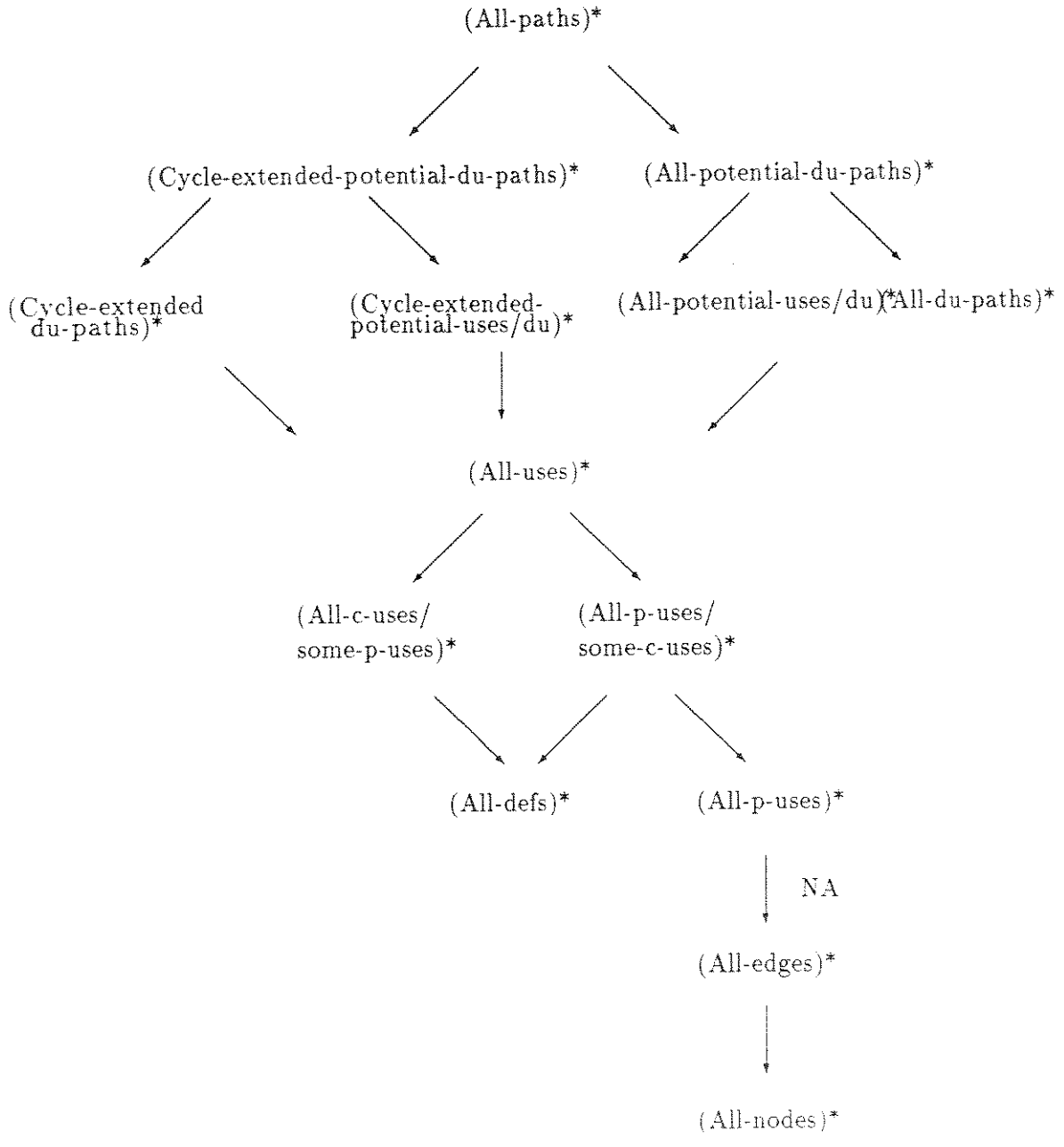


Figura 3.15: Ordem Parcial entre a Família de Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados (DFCF) na Presença de Caminhos não Executáveis Considerando a Propriedade NA—No-Anomalies

minhos, pode-se concluir que todos os critérios Potenciais Usos demandaram $2^t + 2^{t-1}$ casos de teste na presença de caminhos não executáveis e este número é que constitui, na realidade a complexidade dos critérios Potenciais Usos. Esta restrição é de certa forma "equivalente" aos aspectos discutidos em [MAL88b] sobre a inclusão de um caminho π_1 por um caminho π_2.

É importante observar-se que, para o exemplo da Figura 3.17 são requeridos 2^t casos de teste para ambos os critérios todos-usos e todas-definições da FCFD de Rapps e Weyuker e que a mesma consideração na presença de caminhos não executáveis é válida para todos os critérios da FCFD. Estes resultados contradizem os resultados apresentados em [WEY84]; por exemplo, Weyuker determinou que a complexidade dos critérios todos-usos seria $(1/4)(t^2 + 4t + 3)$.

De acordo com Ntafos [NTA88] os critérios "*required pairs*", "*data-contexts*", "$2-dr$ *iterations*" têm a mesma complexidade dos critérios todos-usos; mas todos eles incluem o critério todas-definições e, consequentemente, têm complexidade maior ou igual que a complexidade deste critério.

O critério "*all simple O/I caminhos*", por sua vez, tem complexidade $2^{2t-2} + 2^{t-1}$. Considere a Figura 3.16; é fácil concluir que, com uma distribuição adequada de definições e usos de variáveis, este critério requereria $2^{2t-2} + 2^{t-1}$ *caminhos-OI simples*; neste caso, caminhos completos. Ural [URA88] estabeleceu a complexidade deste critério como 2^t, o que é contraditório com o resultado aqui apresentado.

3.3 Considerações Finais

Neste capítulo foram introduzidas a Família de Critérios Potenciais Usos e a correspondente família de critérios executáveis. Uma característica dessa família de critérios que a distingüe dos demais critérios de teste estrutural baseado em fluxo de dados é que as associações são estabelecidas independentemente da ocorrência explícita de um uso de variável; todos os demais critérios baseados em fluxo de dados exigem a ocorrência de um uso para que um elemento seja requerido. Essa característica foi obtida com a introdução do conceito de *potencial uso*.

Mostrou-se, através da análise de inclusão e do estudo da complexidade, que os critérios Potenciais Usos satisfazem os requisitos básicos exigidos de um bom critério de teste, ou seja, estabelecem uma hierarquia de critérios entre os critérios todos os ramos e todos os caminhos, mesmo na presença de caminhos não executáveis; incluem o critério todas-def; e requerem um número finito de casos de teste. Além disto, mostrou-se que nenhum outro critério de teste baseado em fluxo de dados inclui os critérios Potenciais Usos.

Concluiu-se, também, que todos os critérios baseados em fluxo de dados têm complexidade de ordem exponencial e, portanto, do ponto de vista teórico, estes critérios não seriam aplicáveis na prática. No entanto, estudos empíricos têm demonstrado que, para programas reais, esses critérios demandam um pequeno número de casos de teste. Estudo dessa natureza, descrito no Cap. 6, foi conduzido para os critérios Potenciais Usos Básicos e observou-se que estes critérios seriam, sem dúvida, aplicáveis, a custos aceitáveis, em ambientes comerciais de produção de software.

A complexidade de um critério é um fator importante para as atividades de teste, principalmente na estimativa e determinação de custo associado a essas atividades. Os resultados aqui apresentados, indicam que este tópico, análise de complexidade, deve ser cuidadosamente reanalisado no sentido de contribuir para a caracterização de benchmarks e de diretrizes para a condução de estudos empíricos. Uma vez que todos os critérios têm complexidade maior do que 2^t, a importância desses estudos empíricos, para demonstrar a exeqüibilidade desses critérios, é inquestionável.

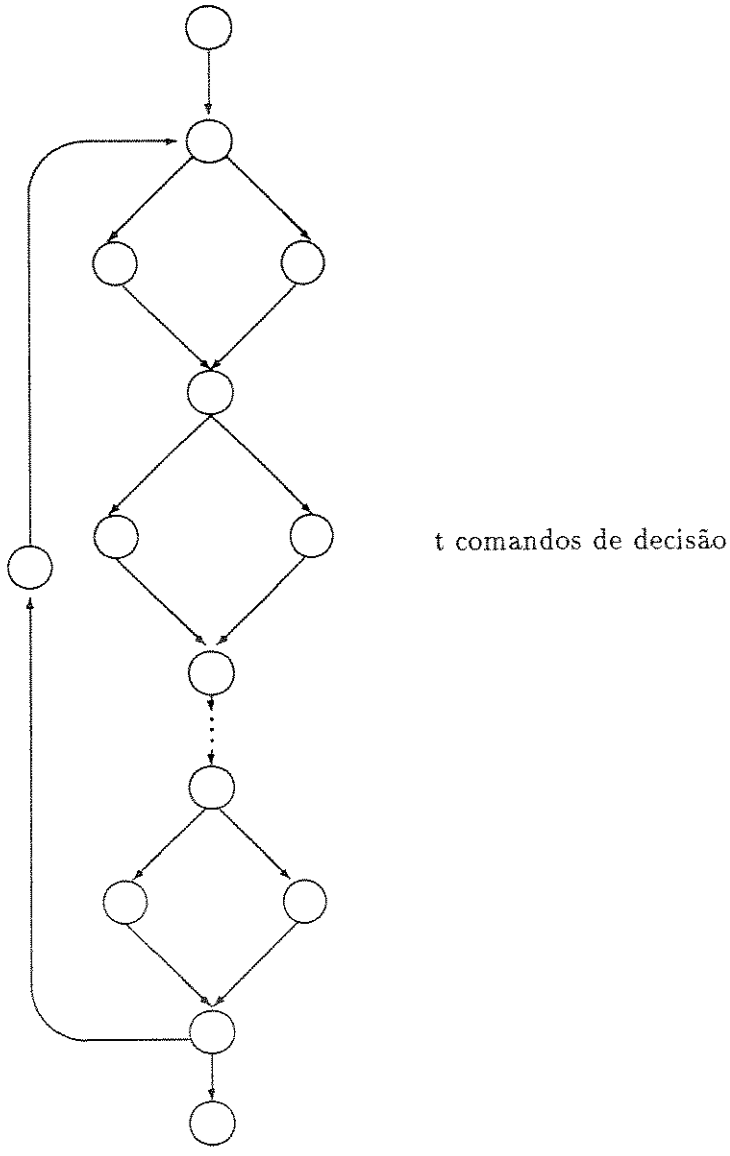


Figura 3.16: Estrutura de Controle que Maximiza o Número de Potenciais-du-caminhos

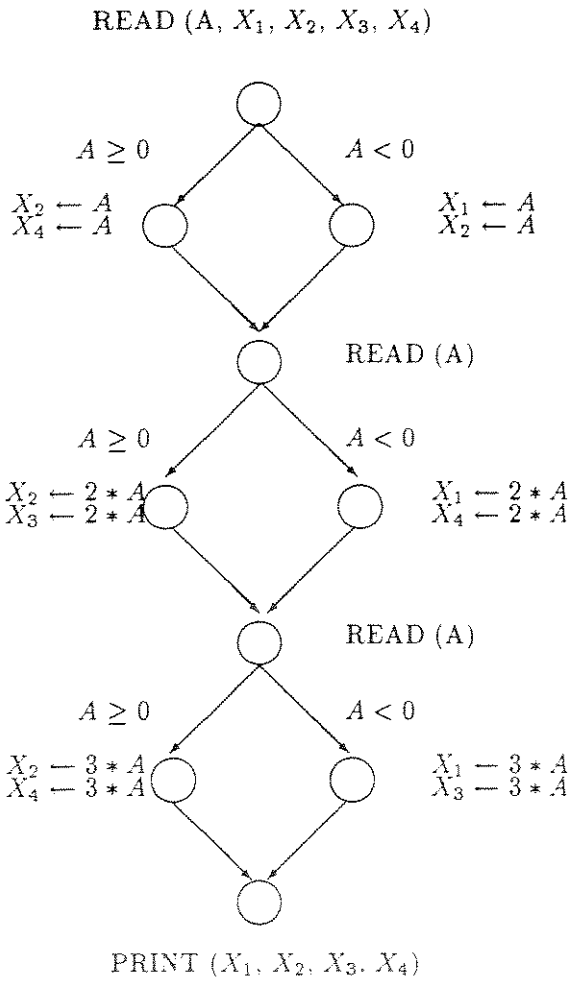


Figura 3.17: Exemplo para Análise de Complexidade

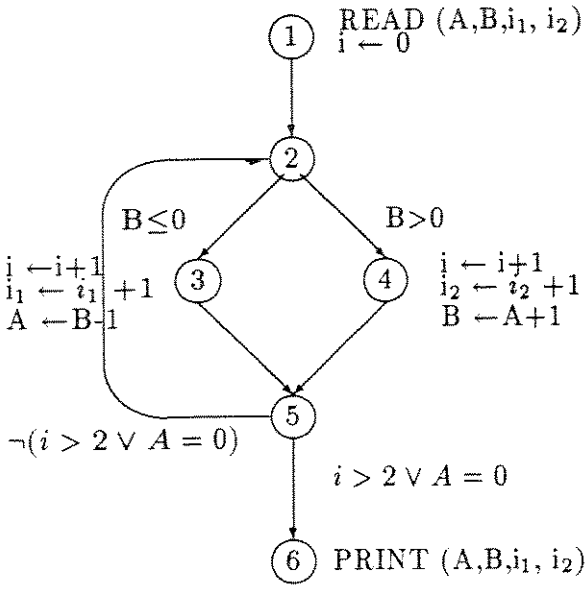


Figura 3.18: Exemplo para Análise de Complexidade na Presença de Caminhos não Executáveis com t=2

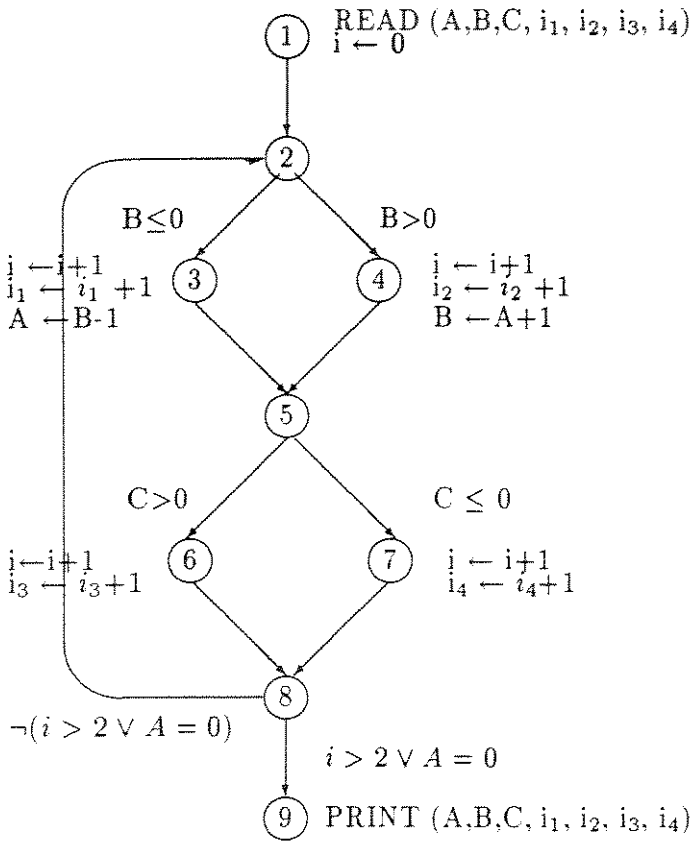


Figura 3.19: Exemplo para Análise de Complexidade na Presença de Caminhos não Executáveis com t=3

Capítulo 4

Modelos de Implementação dos Critérios Potenciais Usos

Neste Capítulo são discutidos alguns aspectos teóricos que nortearam a especificação e o projeto da ferramenta de suporte à aplicação dos Critérios Potenciais Usos. Uma das utilizações de um critério de teste estrutural é na análise de cobertura, ou seja, utilizá-lo como um critério de cobertura; nesse caso, atividades como instrumentação da unidade em teste, monitoração dos caminhos executados pelo conjunto de casos de teste fornecido e determinação dos elementos requeridos executados e não executados, entre outras, devem ser incorporadas à ferramenta.

Considerando que uma das direções de pesquisa de interesse é a combinação de critérios objetivando estabelecer critérios mais "fortes", o que envolve, entre outras, atividades de comparação teórica e empírica entre critérios, e atividades de estudo da adequação desses critérios à classes de erros, procurou-se abstrair alguns conceitos que pudessem caracterizar um conjunto de recursos que facilitassem a incorporação de outros critérios na POKE-TOOL. Em conseqüência caracterizaram-se quatro modelos básicos: *Modelo de Grafo de Fluxo de Controle; Modelo de Instrumentação; Modelo de Dados; e, Modelo de Descrição dos Elementos Requeridos*, discutidos a seguir. O conceito de *arco essencial (arco primitivo)* [CHU87] é explorado para caracterização do Modelo de Descrição do Elementos Requeridos.

4.1 Modelo de Fluxo de Controle

No modelo de fluxo de controle adotado, um programa P é representado por um *grafo dirigido* $G(N, A, s)$, onde os blocos disjuntos de comandos correspondentes da linguagem intermediária LI (ver Apêndice A) são associados aos nós $n \in N$ e os possíveis fluxos de controle entre os blocos são associados aos arcos $e \in A$. O nó s representa o nó de entrada. Este grafo é usualmente denominado *grafo de fluxo de controle* ou de *grafo de programa*. Relembrando, um programa pode ser decomposto em um conjunto de blocos disjuntos com a propriedade que, uma vez executado o primeiro comando do bloco, os demais comandos são também executados na ordem dada; de uma forma mais precisa, um bloco é um conjunto maximal de comandos ordenados $b = \{s_1, s_2, \ldots, s_n\}$, tal que, se $n > 1$, para $i = 2, 3, \ldots, n$, s_i é o único sucessor execucional de s_{i-1} e s_{i-1} é o único predecessor execucional de s_i [RAP82]. No modelo adotado ter-se-á um único nó de entrada e um único nó de saída, o que facilita a aplicação de algoritmos já consagrados da análise de fluxo de dados [HEC77].

As Figuras 4.1, 4.2, 4.3 e 4.4 representam as construções básicas do grafo de fluxo de controle adotadas para os comandos da linguagem LI, a menos dos comandos de desvios incondicionais. Note que, para o comando "case", existe uma possível ligação do nó k para o nó l; esta ligação existe no caso em que não foi especificado um rótulo "default" no comando "case". O grafo de controle de uma unidade é obtido pela simples concatenação dessas construções básicas. Observe-se que o grafo de fluxo de controle da unidade em teste é independente da linguagem de implementação da unidade; no entanto, o usuário configurador da POKE-TOOL deverá levar este modelo em consideração pois, de certa forma, ele reflete aspectos da semântica da LI e terá forte influência na instrumentação da unidade em teste.

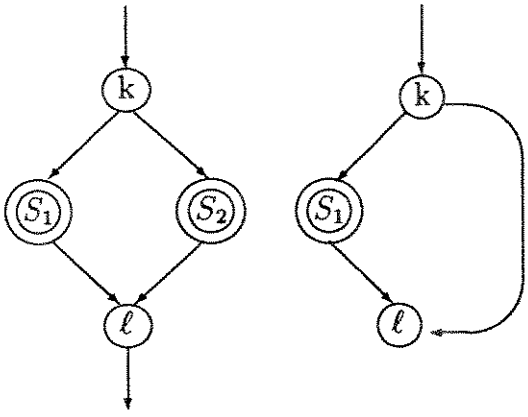
A representação dos comandos de desvios condicionais e incondicionais introduz modificações profundas no fluxo de controle e, consequentemente, no modelo e na instrumentação. O comando de desvio incondicional irrestrito *goto* encerra a caracterização de um bloco, sendo que existirá um arco do bloco que contém o comando *goto* para o bloco que contém o rótulo associado a ele. O uso indisciplinado deste comando pode levar a um grafo de programa onde um ou mais nós poderiam ser agrupados em um único nó; esta situação não afeta os resultados e as análises apresentadas. Os comandos de desvios incondicionais controlados — *break*, *continue* e *return* — também encerram a caracterização de um bloco e existirá um arco do bloco que contém estes comandos para o nó (bloco) que contém o primeiro comando após o comando de iteração ou comando *case* que envolve o nó, no caso do comando *break*; para o nó que representa o encerramento do corpo da iteração, no caso do comando *continue*; ou para o nó de saída, no caso do comando *return*.

Os modelos correspondentes às alternativas de ocorrências dos comandos de desvio com os demais comandos da LI são representados nas Figuras 4.5 e 4.6. Observe-se que alguns nós não alcançáveis podem ser gerados, facilitando a identificação de código não executável, se existir código associado a esses nós; esses nós poderiam facilmente ser

eliminados através de um pós-processamento ou mesmo pela modificação do modelo. Esses nós não introduzem nenhuma complicação para a implementação das outras funções da POKE-TOOL. Atenção especial deve ser dada na instrumentação quanto aos comandos de desvio incondicional como no caso de uma unidade ter mais de um comando *return*.

Comandos de seleção

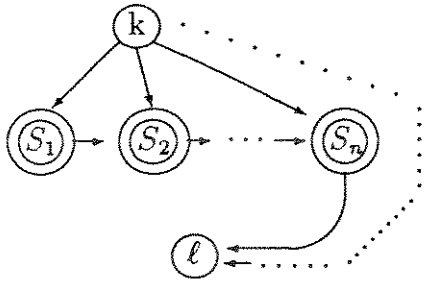
< if > ::= < if-atm >< cond-atm>< statement$_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statment$_2$ >



(1) – representa o subgrafo correspondente ao statement$_1$
(2) – representa o subgrafo correspondente ao statement$_2$.
(k) – associado a cond-atm.

instrumentação: Sejam i e j os nós de entrada dos subgrafos S_1 e S_2, respectivamente. No início dos blocos k, i, j e são inseridas pontas de prova que caracterizam a execução destes blocos, ou seja, pontas de prova k, i, j e .

< CASE >::= < case-atm >< case-cond-atm > { { (< rotc-atm > | < rotd-atm >)
{ < statement > } } }



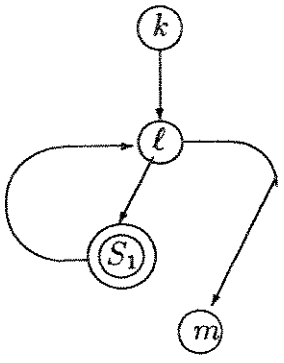
(i) i=1, ..., n representa os subgrafos correspondentes ao statement$_i$ relativos a cada uma das alternativas da seleção múltipla
(k) – associado a case-cond-atm.

instrumentação:
Sejam i_1, i_2,..., i_n os nós de entrada dos subgrafos S_1, S_2,..., S_n, respectivamente. No início dos blocos k, i_1, i_2,..., i_n, l são inseridas pontas de prova que caracterizam a execução destes blocos, ou seja, pontas de prova k, i_1, i_2,..., i_n e l.

Figura 4.1: Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comandos de Seleção.

Comandos de Iteração

< while > ::= < while-atm >< cond-while-atm>< statement>

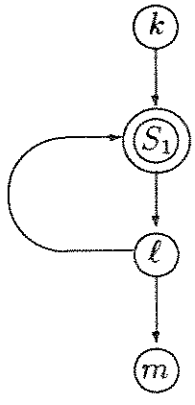


(S_1) – representa o subgrafo correspondente a < *statement* >, ou seja, ao corpo do laço

(ℓ) – associado a < cond-while-atm>

instrumentação: seja i o nó de entrada do subgrafo S_1. No nó k é inserida a ponta de prova k; no nó i as pontas de prova l e i e no nó m, as pontas de prova l e m.

<repeat-until > ::= < repeat-atm >< statement >< until-atm>< cond-until-atm>



(S_1) – representa o subgrafo correspondente a < statement >

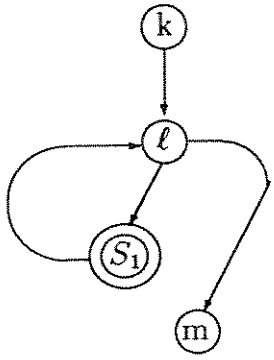
(ℓ) – associado a < cond-until-atm>

instrumentação: seja i o nó de entrada do subgrafo S_1 e j o nó saída de S_1. No início dos nós i, j, k e m são inseridas as pontas de provas i, j, k e m. Adicionalmente, no fim do nó j, é inserida a ponta de prova l.

Figura 4.2: Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comandos de Iteração — "while" e "repeat".

Comandos de Iteração

< for > ::= < for-atm >< S_1 >< cond-for-atm>< S_2 >< statement>



(S_1) – representa o subgrafo correspondente a < *statement* >, ou seja, ao corpo do laço

(k) – associado aos comandos de iniciação da variável de controle do "for", ou seja, aos comandos representados por < S_1 >.

(j) – associado aos comandos que alteram a variável de controle, ou seja, aos comandos representados por < S_2 >, onde o nó j é o nó saída do subgrafo S_1.

(ℓ) – associado a < cond-for-atm>

instrumentação:

sejam i e j os nós de entrada e saída, respectivamente, do subgrafo < S_1 >.
No início do nó k, é inserida a ponta de prova k; no início do nó i as pontas de provas l e i e no início do nó m, as pontas de provas l e m.
No nó j é também inserida a ponta de prova j.

Figura 4.3: Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comando de Iteração "for".

Comandos Sequenciais

< dcl > e < s >

. os comandos sequenciais são associados a um único nó; não geram alternância de fluxo

Comandos de Desvio Incondicional

< goto >

< break >

< continue >

< return >

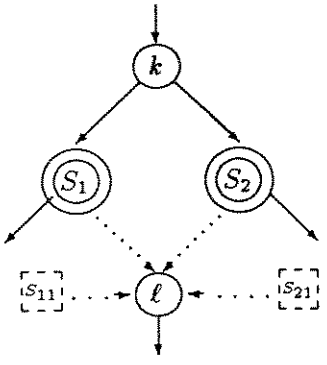
estes comandos são sempre o último comando associado ao bloco i (nó i), ou seja, todos têm a ação de encerrar um bloco; a ação de fluxo de controle destes comandos é melhor entendida e depende do local da ocorrência do comando (ver Figuras 4.5 e 4.6)

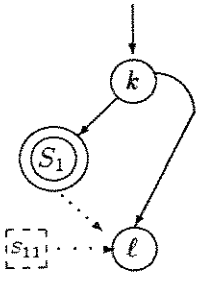
Figura 4.4: Modelo de Fluxo de Controle e Instrumentação Associado aos Comandos da Linguagem LI: Comandos Sequenciais e de Desvio Incondicional.

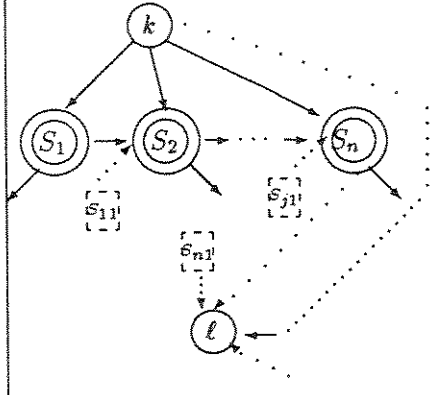
Comandos de Seleção &
Comandos de Desvio Incondicional

<if> ::= < if-atm ><cond-atm>< statement $_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statment$_2$ >.

<case> :=<case-atm><case-cond-atm> {{(<rotc-atm> | <rotd-atm>){ <statement> }}}







– representa o subgrafo correspondente a comandos do início de statement$_i$ até a ocorrência de comandos de desvio incondicional

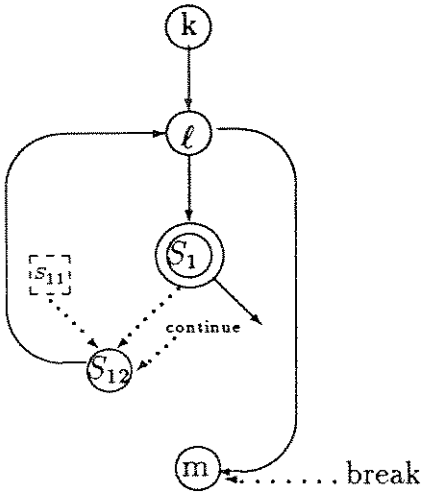
– representa subgrafos correspondentes a comandos após a ocorrência de comandos de desvio incondicional em statement$_i$. Estes blocos podem representar comandos não alcançáveis.

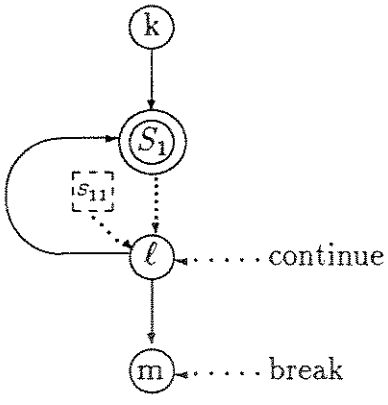
instrumentação: Seja e o nó de saída da unidade em teste. Seja i o nó de entrada do subgrafo S_i e i_1 o nó de entrada do subgrafo S_{i1}. Seja d o nó em que ocorre o comando de desvio incondicional. No nó d é inserida a ponta de prova d e adicionalmente a ponta de prova e se ocorrer o comando "return". O caso em que o comando "if" está aninhado em uma estrutura "while", "for" ou "repeat" é tratado na Figura 4.6. Aos nós i e i_1 são inseridas as pontas de prova i e i_1.

Figura 4.5: Modelo de Fluxo de Controle e Instrumentação Considerando os Comandos de Desvio Incondicional da LI: Comandos de Seleção

Comandos de Iteração &
Comandos de
Desvio Incondicional

<while> ::= < while-atm ><cond-while-atm><statement>
<repeat-until>::=<repeat-atm><statement><until-atm><cond-until-atm>
<for> ::= < for-atm >< S_1 ><cond-for-atm>< $S2$ >< statement>





(S_1) – representa o subgrafo correspondente a comandos do início de < *statement* > até a ocorrência de comandos de desvio incondicional

(S_{11}) – representa o subgrafo correspondente a comandos após a ocorrência de comandos de desvio incondicional em <statement>. Estes blocos podem representar comandos não alcançáveis.

(S_{12}) – representa o fim do corpo do comando de iteração. No caso de comando "for", está associado aos comandos de alteração da variável de controle, representados por < S_2 >.

instrumentação: Seja e o nó de saída da unidade em teste. Seja i o nó de entrada do subgrafo S_1.
Seja d o nó em que ocorre o desvio incondicional. No nó d é inserida a ponta de prova d e adicionalmente a ponta de prova e se ocorrer o comando "return". No caso dos comandos "for" e "while", se ocorrer o comando "break" dentro dessas estruturas, ao nó m são inseridas as pontas de prova l e m; a ponta de prova m é rotulada com outm; No nó d é então inserida a instrução de desvio incondicional para o rótulo outm, na linguagem de implementação da unidade em teste.

Figura 4.6: Modelo de Fluxo de Controle e Instrumentação Considerando os Comandos de Desvios Incondicional da LI: Comandos de Iteração

4.2 Modelo de Instrumentação

A instrumentação visa a oferecer facilidades para a análise posterior à execução dos casos de testes como a análise de adequação de um dado conjunto de casos de teste. Para tanto, este módulo modifica, com inserção de código fonte, a própria unidade em teste, gerando uma nova versão do programa, usualmente denominada *unidade instrumentada*. No caso da POKE-TOOL a unidade instrumentada é armazenada no arquivo *TESTEPROG.C*. O arquivo TESTEPROG.C, do Apêndice B, ilustra a unidade instrumentada para o exemplo ENTAB.C.

A instrumentação consiste essencialmente em inserir pontas de provas nos blocos de comandos correspondentes a cada nó do grafo de programa da unidade em teste, possibilitando a identificação do caminho executado pelo caso de teste fornecido. Uma ponta de prova consiste basicamente em um comando de escrita do número do nó em um arquivo (arquivo *PATH.TES*, na versão atual da POKE-TOOL), produzindo um "trace" do caso de teste fornecido. Esta informação é imprescindível para a função de avaliação da POKE-TOOL. Observe-se no exemplo ENTAB.C (Apêndice B), que várias outras informações são inseridas na unidade instrumentada, facilitando as atividades de teste, assim como outras correlatas, como depuração e manutenção.

A escrita de todos os nós foi adotada para simplificar a implementação, pois, para os modelos adotados de avaliação e de descrição dos caminhos e associações requeridos, seria suficiente inserirem-se pontas de prova nos nós n que contivessem definição de variáveis ou que constituissem arcos primitivos [CHU87] (ver Seção 4.4.1). Além disto, a escrita de todos os nós pode facilitar a implementação de outras funções; por exemplo, a determinação do número de vezes que um determinado comando foi executado.

Obviamente, a instrumentação deve ser tal que reflita a semântica dos comandos da LI e ao mesmo tempo viabilize a correta avaliação dos caminhos efetivamente executados; observe-se também que a instrumentação está fortemente restrita ao modelo de fluxo de controle adotado. Neste sentido cuidados essenciais devem ser tomados quanto aos comandos de: *iteração*, *iteração com break* e *return*. A ocorrência de comandos *break* dentro de comandos *for* e *while* requer a inserção de uma ponta de prova rotulada no bloco de comandos executados logo após o fim da iteração — os rótulos são nomeados *outi*, $i \in N$ — e um comando de desvio incondicional imediatamente antes da ocorrência do comando *break*, na linguagem de implementação da unidade em teste. Em todo bloco que contiver o comando *return* deve-se inserir, além da ponta de prova que caracteriza a execução do bloco, uma ponta de prova com o número do nó saída, a menos que o comando *return* esteja contido no bloco de comandos correspondentes ao nó saída.

As Figuras 4.1, 4.2, 4.3, 4.4 e 4.5 ilustram a instrumentação associada aos comandos da LI nas suas mais diversas possibilidades. Um fato importante a ser observado é que nenhuma modificação é introduzida nos comandos originais do programa fonte, sendo que somente pontas de provas (rotuladas ou não) e comandos de desvios

incondicionais para pontas de provas rotuladas são inseridas. Obviamente, são também inseridos comandos para a abertura e fechamento do arquivo que conterá o caminho executado (arquivo PATH.TES), assim como comandos de declaração e de iniciação das variáveis necessárias para a correta implementação desta função.

A forma de instrumentação acima descrita não é apropriada para procedimentos recursivos pois somente o caminho referente à última chamada da unidade em teste é gravado para cada caso de teste. Frankl [FRA87] propõe uma solução para tratamento de recursividade que consiste em marcar dinamicamente os números dos nós para identificar o nível da recursão e, posteriormente, separar os caminhos efetivamente executados em cada chamada. Nesta proposta fica evidente que as redefinições das variáveis na *i-ésima* chamada seria invisível na $(i-1)-esima$ chamada da unidade em teste para a determinação dos caminhos e associações requeridos. Esta abordagem seria facilmente suportada na POKE-TOOL criando-se um arquivo PATHi.TES para cada chamada e no final da execução do caso de teste, gerar-se-ia um único arquivo PATH.TES, que seria a concatenação de todos os arquivos PATHi.TES. Outra alternativa seria transferir os comandos de manipulação do arquivo PATH.TES para o "driver" da unidade em teste; neste caso, o arquivo PATH.TES conteria a sequência de nós executados porém sem distinção do nível de recursão; esta última alternativa é incompatível com o modelo de avaliação proposto.

Planeja-se incorporar na POKE-TOOL facilidades para análise e testes de procedimentos recursivos e estas alternativas deverão ser analisadas mais cuidadosamente. Pode-se, no entanto, apontar que a primeira abordagem é aparentemente mais compatível com o tratamento dado a chamadas de procedimentos em geral, ou seja, para a determinação de caminhos e associações requeridos; a chamada recursiva seria tratada como a chamada de um procedimento qualquer e do ponto de vista de avaliação seria gerado um "trace" correspondente a cada chamada do procedimento.

4.3 Modelo de Fluxo de Dados

No contexto de teste de software, a análise de fluxo de dados usualmente é utilizada para estender o grafo de programa pela associação de *tipos de ocorrências de variáveis* aos elementos deste grafo que, posteriormente, é utilizado para determinação dos caminhos e associações a serem requeridos. Basicamente, têm-se três tipos de ocorrências de variáveis: *definição, uso e indefinição* . O *uso* de uma variável ocorre quando seu valor é lido da posição de memória correspondente; do ponto de vista dos critérios Potenciais Usos este tipo de ocorrência não precisa ser identificado. Os outros dois tipos de ocorrência são discutidos a seguir.

No caso dos critérios Potenciais Usos é suficiente associar a cada nó i do grafo de programa o conjunto de variáveis definidas no bloco de comandos correspondente; a esta extensão denomina-se *grafo def.*

Para a correta geração do grafo def é necessário precisar o que considera-se uma definição de variável. Uma *definição de variável* ocorre quando um valor é armazenado em uma posição de memória . Em geral, uma ocorrência de variável em um programa é uma definição se ela está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; ou iii) em chamadas de procedimentos como parâmetro de saída. A passagem de valores entre procedimentos através da passagem de parâmetros pode ser feita por: *valor*, *referência* ou *nome* [GHE87]. Se a variável for passada por referência ou por nome considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são distinguidas das demais e são ditas *definidas por referência*; esta distinção é utilizada na geração dos grafo(i) (ver seção 4.4.2), ou seja, na determinação dos caminhos livres de definição para as variáveis definidas em i. Resumidamente, uma variável v_1 definida por referência faz parte do conjunto de variáveis definidas em i, ou seja, $v_1 \in defg(i)$, para construção dos grafo(i), mas qualquer definição por referência de v_1 em um nó $j \neq i$ não é considerada como redefinição de v_1; este enfoque é conservador no sentido de que não deixa de requerer nenhum fluxo de dados que possa existir, ou seja, a seleção de caminhos e associações é mais rigorosa.

Uma complicação é introduzida quanto ao tratamento de ponteiros e de variáveis compostas — variáveis estruturadas (vetores e matrizes) e registros —, pois não é possível, em geral, determinar-se estaticamente o elemento particular dessas variáveis que seria referenciado. Para variáveis estruturadas adotou-se que a definição de um elemento da variável implica a definição da variável; por exemplo, se A é um vetor e ocorre a definição de $A[e]$, onde e é uma expressão, então é considerado que ocorreu a definição de A. Para variáveis do tipo registro adotou-se a mesma abordagem, ou seja, qualquer definição de um campo de um registro é considerado definição do registro. Variáveis do tipo ponteiro são tratadas exatamente como uma variável comum e não é feito o tratamento de definições por *dereferenciação.* Outra vez, não foram tratados esses tipos de definição porque é, em geral, impossível saber estaticamente que objeto de dado [GHE87] estaria sendo definido por dereferenciação. Observe-se

que o enfoque adotado para variáveis compostas não é conservador, pois alguns fluxos de dados existentes podem não ser requeridos. Um enfoque conservador, semelhante ao adotado para as variáveis definidas por referência, poderia ser adotado para variáveis compostas. Adicionalmente, pretende-se estender este modelo para tratar definições por dereferenciação; uma abordagem usualmente adotada em análise estática [POD90] é considerar um acesso através de um ponteiro como um acesso a todos os objetos que possam ser apontados pelo ponteiro. Pretende-se, adicionalmente, conduzir-se estudos comparativos destas diferentes abordagens.

No modelo de fluxo de dados adotado, considera-se que no nó de entrada ocorre uma definição dos parâmetros e das variáveis globais que ocorrem na unidade em teste.

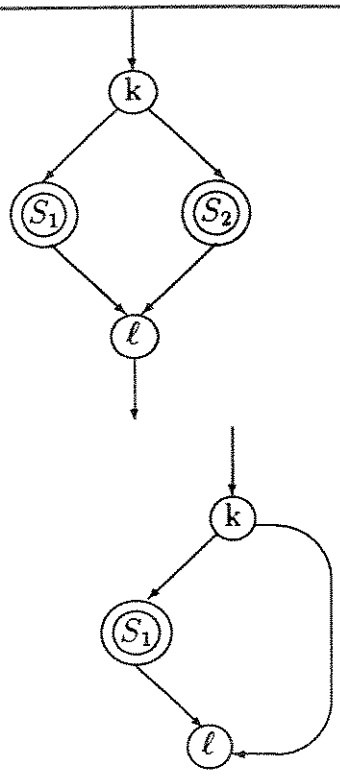
Ocorre uma *indefinição de variável* se a sua localização não estiver definida ('amarrada') na memória ou se não se tiver acesso ao seu valor [FRA87]. A indefinição de uma variável pode ocorrer devido ao encerramento da execução da unidade; neste sentido o nó de saída tem uma indefinição de todas as variáveis locais.

Outro fator que pode levar à ocorrência de indefinição de variáveis é a permissão de declaração de variáveis no início de blocos; em geral, as regras de escopo utilizadas determinam que o escopo das variáveis é limitado ao bloco onde as variáveis são declaradas, como nas linguagens C e Algol 68 e, portanto, estariam indefinidas fora do bloco onde são definidas. Este aspecto é muito importante na construção dos grafo(i), pois a seleção de caminhos ou associações deve restringir-se ao escopo da variável. O aninhamento de blocos com a ocorrência de declarações de variáveis com o mesmo nome constitui um outro ponto importante. Considere uma variável v declarada em um bloco m, definida no nó i deste mesmo bloco e redeclarada em um bloco mais interno n. Denote-se por v_m e v_n para diferenciar entre a variável v do bloco m e a do bloco n, respectivamente. Pelas regras de escopo mais usuais, a variável v_m não seria visível no bloco n, ou seja, ela é indefinida neste bloco e, portanto, jamais seria redefinida ou utilizada até o encerramento da execução dele. Observe-se ainda que qualquer definição de v_n no bloco n seria invisível para v_m. Após a execução do bloco n a variável v_m seria novamente visível e passível de definição e uso. Consequentemente, a ocorrência de declarações de variáveis em blocos mais internos não deve inibir (afetar) a determinação de caminhos ou associações requeridas.

As Figuras 4.7, 4.8 e 4.9 sintetizam os conceitos e hipóteses básicos para a determinação do grafo def a partir do modelo de fluxo de controle dos principais comandos da linguagem LI. Observe-se que para os comandos sequenciais a extensão é óbvia, uma vez que estes comandos estão sempre associados a um único nó. A generalização destas idéias para a inclusão de comandos de desvios condicionais e incondicionais é simples e direta.

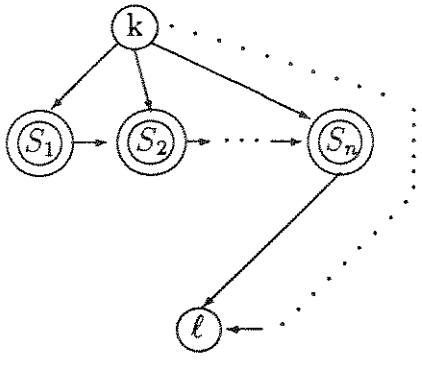
Comandos de Seleção

< if > ::= < if-atm >< cond-atm>< statement$_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statement$_2$ >



Ao nó k são atribuídos o conjunto de variáveis definidas no bloco de comandos associado a este nó e o conjunto de variáveis definidas na condição associada a <cond-atm>. Ao nó l é atribuído o conjunto de variáveis definidas no bloco de comandos associado a este nó. A extensão dos subgrafos S_1 e S_2 depende dos comandos associados a < statement$_1$ > e < statement$_2$ >, respectivamente.

<case>::=<case-atm>< case-cond-atm>{{ (<rotc-atm>| <rotd-atm>) {<statement>} }}

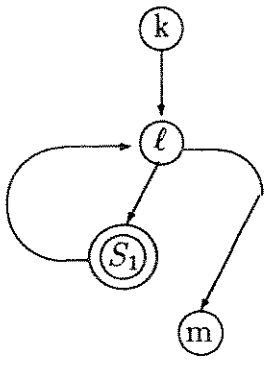


Ao nó k são atribuídos o conjunto de variáveis definidas no bloco de comandos associados a este nó e o conjunto de variáveis definidas na condição do case associada a < case-cond-atm >. Ao nó l é atribuído o conjunto de variáveis definidas no bloco de comandos associado a a este nó. A extensão dos subgrafos S_n depende dos comandos associados a cada uma das alternativas da seleção múltipla.

Figura 4.7: Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Seleção.

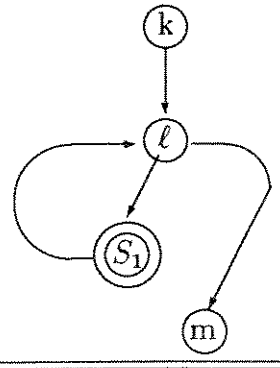
Comandos de iteração

< while > ::≡ < while-atm >< cond-while-atm>< statement>



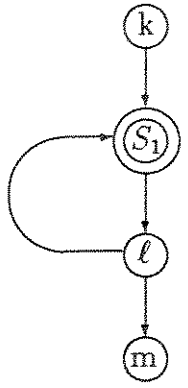
Ao nó k é associada nenhuma definição de variável devido ao comando while , somente devido aos demais comandos eventualmente associados ao nó k. Ao nó é associado conjunto de variáveis definidas na condição associada a cond-while-atm . Ao nó m somente é atribuído o conjunto de variáveis definidas devido a outros comandos associados a este nó. A extensão do subgrafo S_1 depende dos comandos do corpo do laço, ou seja, associados a statement .

< for > ::≡ < for-atm >< S_1 ><cond-for-atm>< S_2 >< statement>



Seja x o nó de saída de S_1. Ao nó k é associado o conjunto de variáveis definidas pelos comandos representados por S_1 . Aos nós e m idem ao comando while . Ao nó x é atribuído o conjunto de variáveis definidas pelos comandos representados por S_2 e, se for o caso, o conjunto de variáveis definidas por outros comandos associados ao nó x. A extensão do subgrafo S_1 depende dos comandos do corpo do laço associados a statement.

<repeat-until> ::≡ <repeat-until><statement>< until-atm><cond-until-atm>



Ao nó k não é associada nenhuma definição de variável devido ao comando repeat-until , somente devido aos demais comandos eventualmente associados ao nó k. Ao nó l é associado o conjunto de variáveis definidas na condição associada a <cond-until-atm>. Ao nó m, somente é atribuído o conjunto de variáveis definidas devido a outros comandos eventualmente associados a este nó. A extensão do subgrafo S_1 depende dos comandos do corpo do laço, ou seja, associados a < statement >.

Figura 4.8: Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos de Iteração.

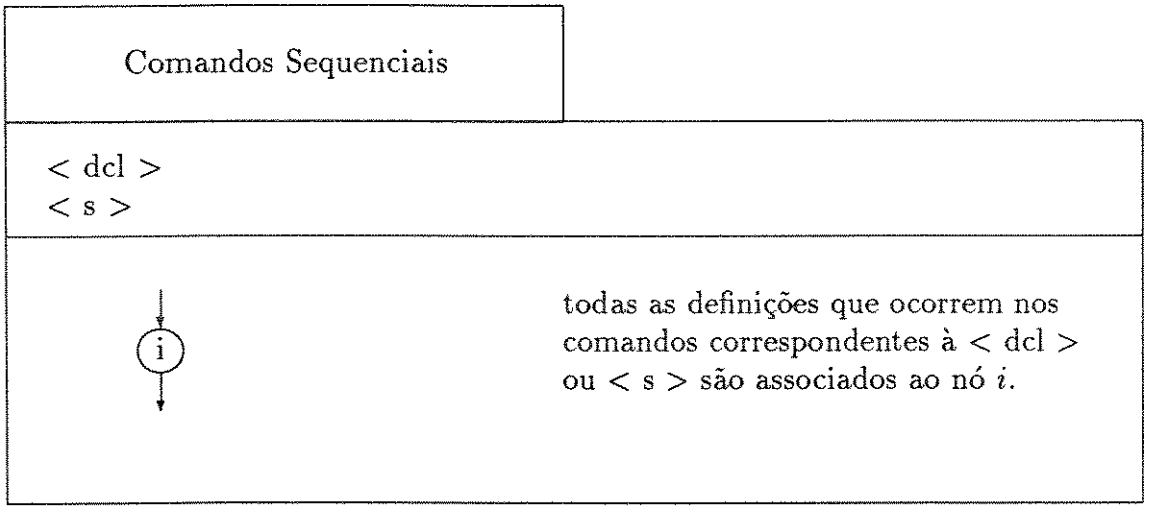


Figura 4.9: Diretrizes para a Expansão do Grafo de Programa para Obtenção do Grafo def: Comandos Sequenciais.

4.4 Modelo de Descrição dos Elementos Requeridos

Nesta seção é apresentado o modelo de descrição dos elementos requeridos pelos Critérios Potenciais Usos; este modelo é fundamentado no conceito de arco primitivo [CHU87] e no conceito de Grafo(i) [MAL88b]. Com a utilização destes conceitos espera-se fornecer um núcleo básico para o suporte à aplicação de critérios de teste estrutural, de uma maneira geral. A seguir, discute-se o algoritmo proposto por Chusho — *Algoritmo de Redução de Herdeiros* — para obter, a partir de um grafo de fluxo de controle, um grafo que contém somente arcos primitivos, denominado *grafo com redução de herdeiros.* As modifições necessárias introduzidas no algoritmo de Chusho para viabilizar o uso desses conceitos no contexto de fluxo de dados são apresentadas. O algoritmo modificado è denominado *Algoritmo de Redução de Herdeiros de Fluxo de Dados.* Posteriormente, discutem-se as diretrizes para a caracterização e descrição dos caminhos e associações requeridos pelos critérios Potenciais Usos a partir do conceito de grafo(i).

4.4.1 Utilização de Arcos Primitivos no Contexto de Fluxo de Dados

Discute-se a seguir, o uso do conceito de arco primitivo, introduzido por Chusho [CHU87], dentro do contexto de teste estrutural baseado em análise de fluxo de dados, para viabilizar a descrição de elementos requeridos por critérios de teste dessa natureza.

O conceito de arco primitivo se baseia no fato de existirem arcos dentro de um grafo de fluxo de controle (GFC) que são sempre executados quando um outro arco é executado; esses arcos são ditos não essenciais para a análise de cobertura.

Um arco que sempre é executado quando se executa um outro arco é denotado por *arco herdeiro.* Mais formalmente, se todo caminho completo que inclui o arco a sempre incluir o arco b, então b é chamado *arco herdeiro* de a e a é chamado *arco ancestral* de b, pois b herda informação sobre a execução de a. O conceito de *arco primitivo* é então estabelecido em função do conceito de arco herdeiro, sendo entendido como *arco primitivo* todo arco que *não* é herdeiro de nenhum outro.

Para ilustrar considere o exemplo de [CHU87] reproduzido na Figura 4.10. Observe-se que o arco a é herdeiro de b e c, isto é, sempre que forem executados b ou c, também é executado a. Note-se também que b é sempre executado quando são executados d ou e, logo b, é herdeiro de d e e e, portanto, não é primitivo.

Chusho propõe um algoritmo para redução de um GFC para um grafo onde existem somente arcos primitivos; esse grafo é chamado *grafo com redução de herdeiros.* Esse algoritmo consiste na aplicação de regras para a eliminação dos arcos herdeiros até chegar-se a um grafo onde todos os arcos herdeiros foram eliminados. Todos os arcos do grafo reduzido correspondem a arcos essenciais, ou seja, constituem os arcos primitivos do GFC.

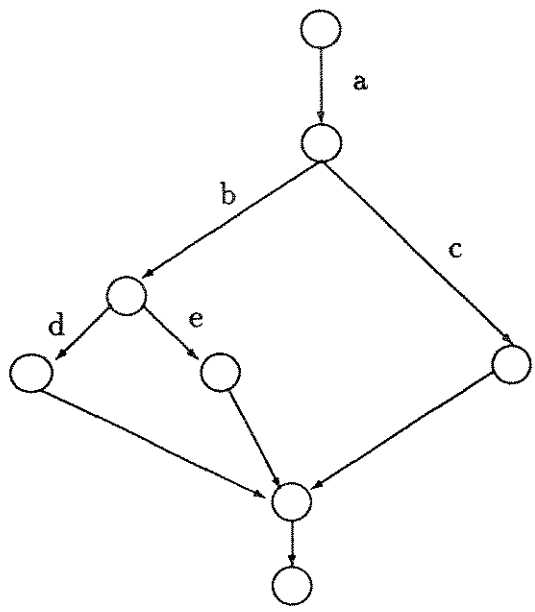


Figura 4.10: GFC com seus Arcos Primitivos e Herdeiros.

Para poder aplicar o conceito de arcos primitivos em critérios baseados na análise de fluxo de dados foi necessário alterar ligeiramente o algoritmo proposto por Chusho. Basicamente, essa alteração consiste em não aplicar uma das regras de redução de herdeiros de Chusho.

A seguir, serão apresentados conceitos e definições introduzidos por Chusho [CHU87], necessários para entendimento do algoritmo de Chusho e do algoritmo modificado. Alguns aspectos da implementação do algoritmo modificado, na POKE-TOOL, são apresentados na Seção 5.2 do Capítulo 5.

Definição 1 Para todo nó x, seja $IN(x)$ o número de arcos entrando em x e $OUT(x)$ o número de arcos saindo de x. Um nó com $IN(x) = 0$ é chamado de *nó entrada*, e x com $OUT(x) = 0$ é chamado de *nó saída*.

Definição 2 Para todo caminho de um nó de entrada para um nó saída, se o caminho incluindo um arco a sempre incluir outro arco b, b é chamado de *herdeiro* de a e a é chamado um *ancestral* de b.

Definição 3 Um arco que nunca é herdeiro de outro arco é chamado um *arco primitivo*.

Definição 4 Para um nó x, um arco (x,x) é denominado auto-laço ("self-loop").

Definição 5 Arcos incidentes no mesmo nó em um caminho são denominados *arcos adjacentes.*

Definição 6 Um grafo dirigido sem herdeiros é chamado um *grafo reduzido de herdeiros.*

Definição 7 Um nó y é chamado um *dominador* de um nó x se todos os caminhos do nó de entrada para x incluem y. Um nó z é chamado um *dominador inverso* de x se todos os caminhos de x para o nó saída incluem z. Sejam *DOM(x)* e *IDOM(x)* os conjuntos de *dominadores* e *dominadores inversos* de x, respectivamente.

Alguns algoritmos para a obtenção de *DOM(x)* estão contidos em [HEC77, HOR87]; para obter *IDOM(x)* basta aplicar o mesmo algoritmo dos dominadores só que para um grafo cujas direções dos arcos originais foram invertidas.

Teorema 2 Um auto-laço é um arco primitivo.

Teorema 3 Se existe uma relação de herança entre dois arcos que não são adjacentes, o arco herdeiro tem seu arco adjacente como outro ancestral.

Segundo Chusho, a partir dos Teoremas acima, é suficiente considerar se um arco entre diferentes nós é herdeiro de seu arco adjacente ou não. Considere a Figura 4.11 extraída de [CHU87], onde as linhas tracejadas representam um ou mais arcos que podem existir.

A condição para que o arco a seja um herdeiro de b, c, d ou e é examinada a seguir, considerando-se 4 casos distintos:

Caso 1) a é um herdeiro de b: um caminho passando pelo arco b passa necessariamente por a ou c porque x não é um nó de saída; então passa necessariamente por a somente se uma das seguintes condições for satisfeita:

i) não existem arcos c, ou

ii) existem um ou mais arcos c e um caminho passando através de c necessariamente retorna para x, isto é, x é um dominador inverso do nó dreno do arco c.

Caso 2) a é um herdeiro de c: a condição para este caso é idêntica à segunda condição do caso 1.

Caso 3) a é um herdeiro de d: Um caminho passando pelo arco d passa necessariamente palos arcos a ou e, pois o nó y não é um nó de entrada; portanto, passa necessariamente pelo arco a somente se uma das condições abaixo for satisfeita:

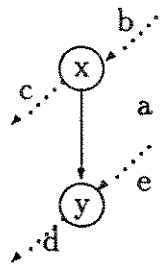


Figura 4.11: Forma Geral de um Arco e seus Dois Nós.

i) não existem arcos e, ou

ii) existem um ou mais arcos e e um caminho passando pelo arco e passa necessariamente por y, isto é, y é um dominador do nó fonte para o arco e.

Caso 4) a é um herdeiro de e: a condição para este caso é idêntica à segunda condição do caso 3.

Os conceitos acima dão origem às quatro regras de redução propostas por Chusho para eliminar os arcos herdeiros de um GFC.

Condição 1 Para um GFC *G(N,A,s)*,

$$x, y \in N \wedge x \neq y \wedge (x, y) \in \mathrm{A}.$$

Regra de Redução R 1 Sob a condição 1, se

$$IN(x) \neq 0 \wedge OUT(x) = 1,$$

(x,y) é eliminado de A e x e y são unidos em um único nó, como mostrado na figura 4.12a.

Regra de Redução R 2 Sob a condição 1, se

$$IN(y) = 1 \wedge OUT(y) \neq 0,$$

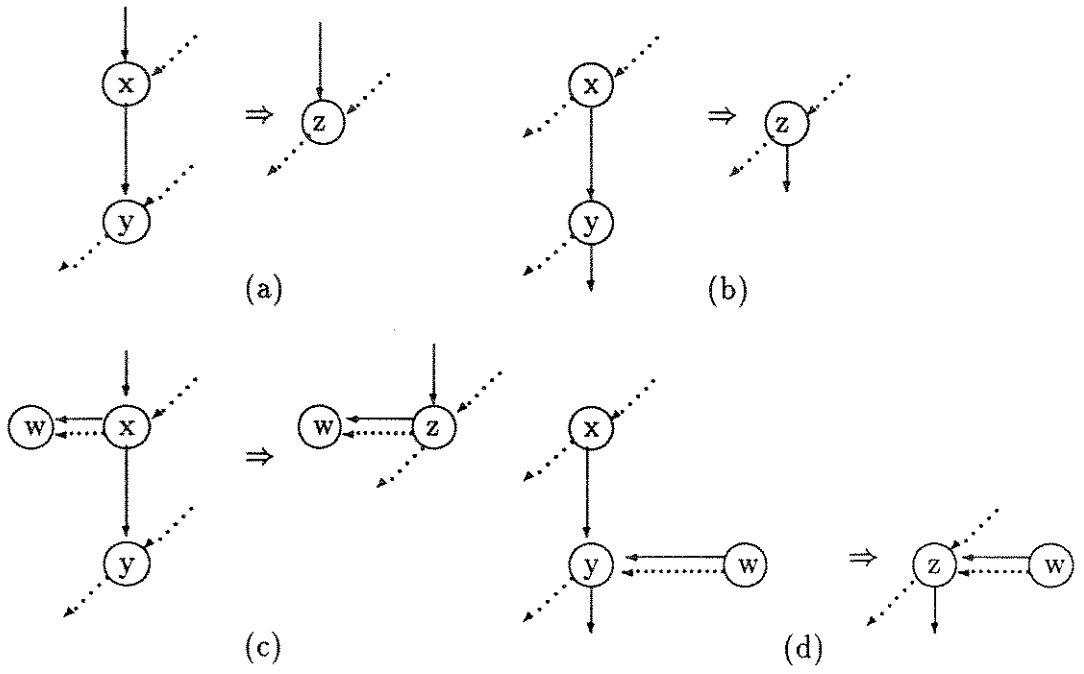


Figura 4.12: Regras para Redução de Herdeiros.

(x,y) é eliminado de A e x e y são unidos em um único nó, como apresentado na Figura 4.12b.

Regra de Redução R 3 Sob a condição 1, se

$$OUT(x) \geq 2$$

e

$$x \in IDOM(w) para \forall w \in \{w \mid (x, w) \in \mathrm{A} \wedge \mathrm{w} \neq \mathrm{y}\},$$

(x,y) é eliminado de A e x e y são unidos em um único nó, como apresentado na Figura 4.12c.

Regra de Redução R 4 Sob a condição 1, se

$$IN(y) \geq 2$$

e

$$x \in DOM(w) \forall w \in \{w \mid (w, y) \in A \wedge w \neq x\},$$

(x,y) é eliminado de A e x e y são unidos em um único nó, como apresentado na Figura 4.12d.

Usando as quatro regras de redução, o algoritmo para transformar um grafo dirigido para um grafo reduzido de herdeiros é dado a seguir:

Algoritmo de Redução de Herdeiros:

1) Aplique **R1** para qualquer arco que satisfaça as condições de **R1**.

2) O passo 1) é repetido até que nenhum arco que satisfaça as condições de **R1** seja encontrado.

3) Aplique **R2** para qualquer arco que satisfaça as condições de **R2**.

4) O passo 3) é repetido até que nenhum arco que satisfaça as condições de **R2** seja encontrado.

5) Escreva uma marca de herdeiro em qualquer arco *(x,y)* que satisfaça as condições de **R3**, se houver pelo menos um arco sem uma marca de herdeiro entre os arcos que entram em x ou entre os arcos que compõem um caminho que parte dos arcos que saem de x para x, exceto o próprio arco *(x,y)*.

6) O passo 5) é repetido até que nenhum arco que satisfaça as condições descritas no passo 5) seja encontrado.

7) Escreva um marca de herdeiro em qualquer arco *(x,y)* que satisfaça as condições de **R4**, se houver pelo menos um arco sem uma marca de herdeiro entre os arcos de saída de y ou entre os arcos que compõem um caminho que chega inversamente a partir dos arcos de entrada de y para y, exceto o próprio arco *(x,y)* .

8) O passo 7) é repetido até que nenhum arco que satisfaça as condições descritas no passo 7) seja encontrado.

9) Elimine qualquer arco com uma marca de herdeiro e junte em um único nó os nós que compõem esse arco.

10) O passo 9) é repetido até que nenhum arco com marca de herdeiro seja encontrado.

A utilização do algoritmo de redução de herdeiros de Chusho no contexto de testes estruturais baseado em análise de fluxo de dados não é direta, pois os arcos primitivos determinados por este algoritmo, uma vez executados, garantem a execução de todos os arcos do GFC e nada garantem a respeito da seqüência em que estes arcos foram executados.

Os critérios baseados em análise de fluxo de dados requerem, em geral, seqüências particulares de arcos; desta forma, para utilizar-se o conceito de arco primitivo para a caracterização de caminhos e associações requeridos por estes critérios, faz-se necessária uma alteração no algoritmo proposto por Chusho que consiste em excluir os passos 5 e 6 do algoritmo original, o que equivale à não aplicação da regra **R3**.

Esse novo algoritmo é denominado *Algoritmo de Redução de Herdeiros de Fluxo de Dados* (REHFLUXDA). O arco ao qual seria aplicada a regra **R3** denomina-se

Arco Primitivo de Fluxo de Dados (pfd). A aplicação da regra **R3** impossibilitaria a descrição de caminhos livres de laços requeridos pelos critérios de fluxos de dados, em particular pelos critérios todos-potenciais-usos/du e todos-potenciais-du-caminhos. Também não seria possível descrever genericamente uma particular associação de forma que todos os caminhos que a exercitassem fossem completamente identificados. Explicitando melhor, considere a Figura 4.12c, que representa um subgrafo de um GFC qualquer, onde pela aplicação da regra **R3**, o arco *(x,y)* seria eliminado.

Seja $N_w = \{w \mid w$ está incluído em algum caminho $\pi = (x, w_1, w_2, \ldots, x)\}$, $N_a = \{n \mid n \in$ antecessores de x e $n \notin N_w\}$ e $N_y = \{m \mid m \in$ sucessores de $y\}$. Seja um *grafo def* tal que existe um nó $n_i \in N_a \mid defg(n_i) \neq \phi$ e existem caminhos livres de definição de n_i para o nó y. Observe-se que para o caso do critério todos os ramos, a execução do arco *(x,y)* é garantida requerendo-se a execução de um arco primitivo (w_1, w_2) onde w_1 e $w_2 \in N_w$. No entanto, no caso dos critérios baseados em fluxo de dados é fácil concluir que, se aplicado a regra **R3**, não seria possível descrever, somente através de arcos primitivos, caminhos livres de laços que envolvessem o arco *(x,y)*. Para o caso de associações, suponha que nos nós $w \in N_w$ não ocorra redefinição das variáveis definidas em n_i; a associação $[n_i, (x,y), defg(n_i)]$ pode ser satisfeita executando-se um caminho $\pi_i = (n_i, n_{i+j}, \ldots, n_{i+n}, x, y)$, onde $n_{i+j} \in N_a, j = 1, \ldots, n$. No entanto, não seria possível, se aplicado a regra **R3**, descrever esta associação somente com arcos primitivos de forma a incluir o caminho π_i.

Ainda para caminhos livres de laços $\pi = (n_i, \ldots, x, y, \ldots, m_1, m_2)$ onde m_1 e $m_2 \in N_y$, também não seria possível descrevê-los somente através de arcos primitivos.

O caso em que o nó y é o nó saída $(out(x) = 0)$ está incluído na análise acima.

Para as demais regras conclui-se facilmente que nenhuma informação de fluxo de dados é perdida. Por exemplo, considere-se a a Figura 4.12d; mesmo se eliminado o arco (x,y) pela aplicação da regra **R4**, qualquer associação ou caminho que envolva o nó n_i e o arco (x,y) será automaticamente requerida ou satisfeita por associações que envolvam um arco primitivo (m_1, m_2) onde $m_1, m_2 \in N_m = \{m \mid m$ está incluído em algum caminho $\pi = (y, \ldots, y)\}$. Se ocorrer a redefinição de todas as variáveis $v \in defg(i)$ no nó y para um determinado caminho $\pi = (n_i, \ldots, y)$, o arco (x,y) é considerado um arco primitivo especial p_ϕ naquele caminho específico, no momento da geração dos grafo(i); da mesma forma se ocorrer a redefinição no nó m_1, o arco (m_0, m_1) é estabelecido como p_ϕ, onde m_0 é o antecessor imediato de m_1 naquele caminho.

Exemplificando, considere a Figura 4.13. Se aplicasse-se a regra **R3** o caminho $\pi = (1,2,4,5,6,7,8,9)$ não teria primitivo associado a ele, no entanto, a execução dos primitivos p_1, p_2 e p_3 garante a execução de todos os arcos do grafo em questão.

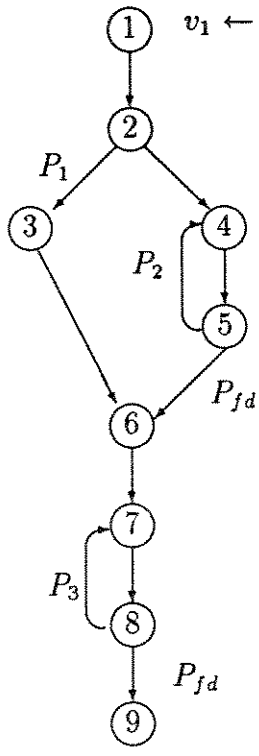


Figura 4.13: Exemplo para Eliminação da Regra R3.

A seguir é apresentado o algoritmo REHFLUXDA.

Algoritmo de Redução de Herdeiros de Fluxo Dados (REHFLUXDA)

```
WHILE existe um arco (x,y) que satisfaz R1
  aplique  R1 no arco (x,y)

WHILE existe um arco (x,y) que satisfaz R2
  aplique R2 no arco (x,y)

WHILE existe um arco (x,y) que satisfaz R4
  IF existe pelo menos um arco sem marca de herdeiro
     entre os arcos de saida de y ou entre os arcos
     que compoem um caminho que chega inversamente em
     y a partir dos arcos de entrada de y, exceto o
     proprio arco (x,y)
  THEN
    Escreva uma marca de herdeiro no arco (x,y)

WHILE existir um arco (x,y) com uma marca de herdeiro
  elimine o arco (x,y) e junte em um unico no os nos x e y
```

O algoritmo REHFLUXDA tem como entrada um grafo de fluxo de controle GFC e retorna como saída o conjunto de arcos primitivos de fluxo de dados desse GFC. Lembre-se que estes arcos primitivos incluem aqueles arcos que seriam eventualmente eliminados pela aplicação da regra **R3**, ou seja, os arcos primitivos de fluxo de dados. A regra **R4** é a mais sofisticada e é a que mais exige em termos de processamento. Para aplicar essa regra, é necessário que seja calculado o conjunto de dominadores *DOM(x)* para todo nó x do GFC. Para calcular esses conjuntos foi utilizado um algoritmo genérico contido em [HEC77]. Detalhes da implementação deste algoritmo são apresentados em [CHA91a].

4.4.2 Grafo(i) e Descritores

Com o objetivo de dar-se uniformidade à implementação de ferramentas de suporte a testes estruturais, em particular aos critérios Potenciais Usos, foi sugerido o conceito de *grafo(i)* [MAL88b], obtido a partir do grafo def e que fornece "todos" os caminhos livres de definição c.r.a. qualquer variável definida em i para todo nó e todo arco alcançável a partir do nó i. A proposição é construir-se um único *grafo(i)* para cada nó $i \mid defg(i) \neq \phi$, 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. A Figura 4.14 contém o grafo(1) e o grafo(5) para o exemplo do Apêndice C.

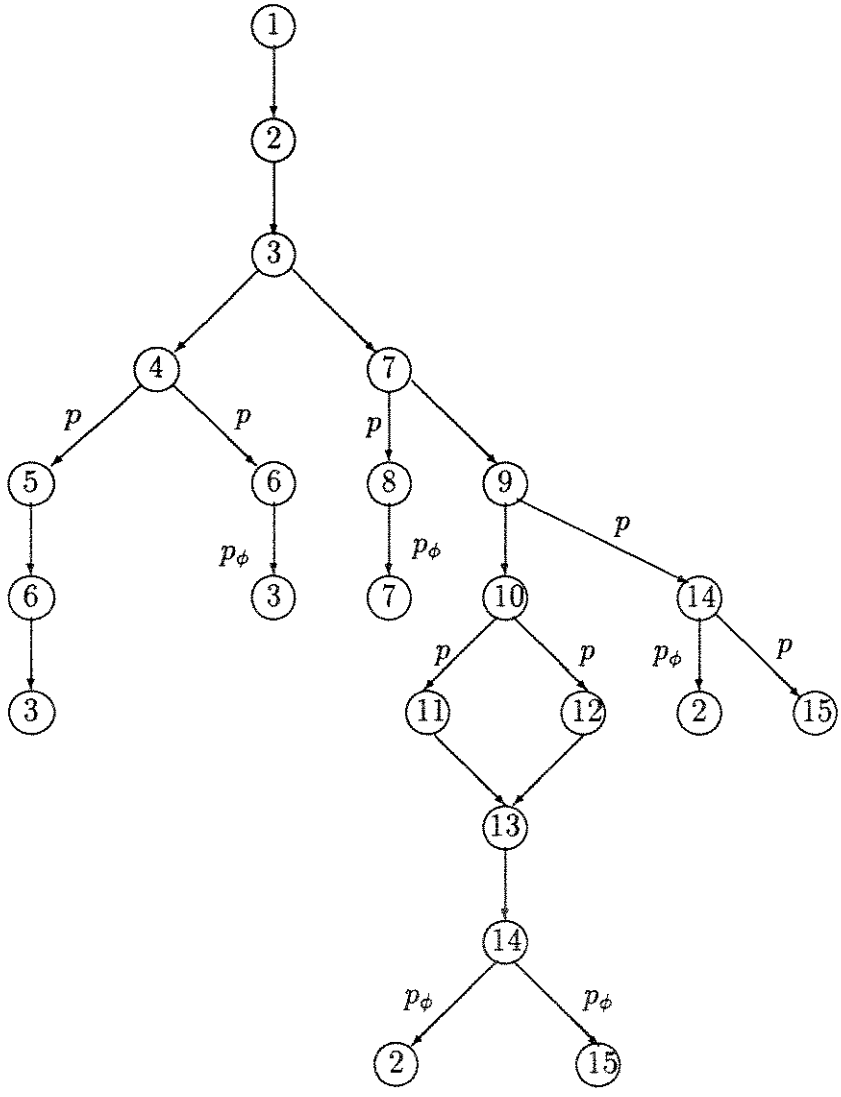
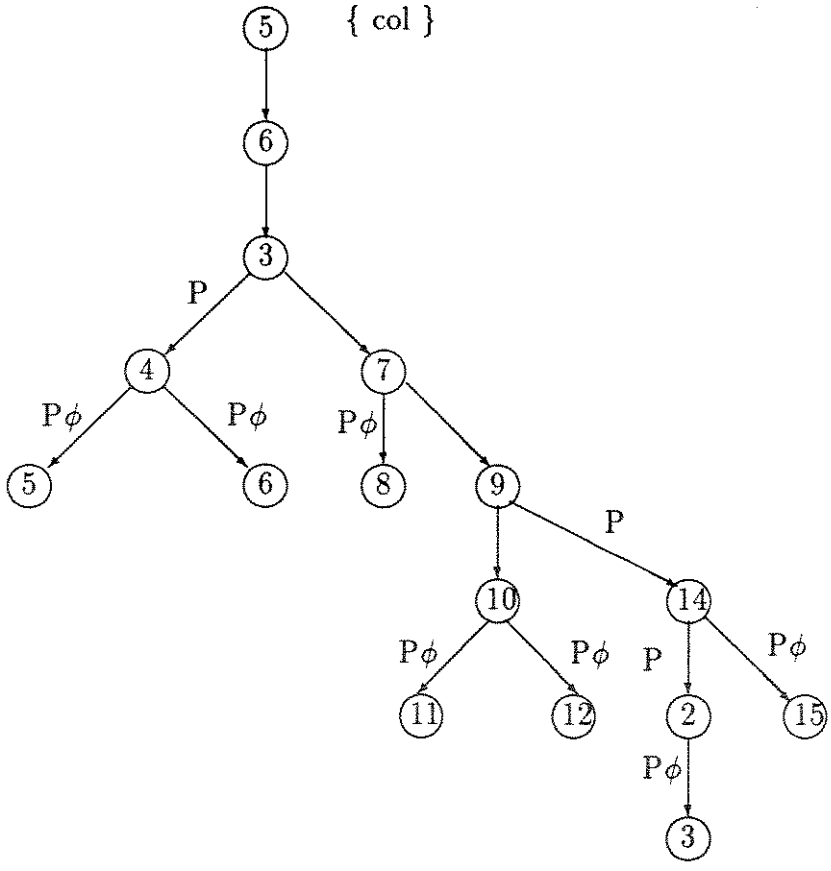


Figura 4.14: Grafo(1) do Exemplo do Apêndice C



15

Figura 4.15: Grafo(5) do Exemplo do Apêndice C.

A partir dos grafo(i) e do conjunto de arcos primitivos estabelecem-se as diretrizes para a caracterização e descrição dos caminhos e associações requeridos pelos critérios Potenciais Usos. Estas informações podem ser utilizadas tanto para a geração de casos de teste utilizando-se, por exemplo, as descrições obtidas como entrada para ferramentas de execução simbólica, como para a avaliação da adequação de um dado conjunto de casos de teste, utilizando-se as descrições como entrada para um módulo avaliador que determinaria, em função dos caminhos efetivamente executados, aqueles caminhos e associações requeridos mas não executados.

O algoritmo para a obtenção dos grafo(i) foi introduzido inicialmente em [MAL88b] e encontra-se detalhado em [CHA91a, CHA91b]. Esse algoritmo consiste essencialmente em uma busca em profundidade no grafo def a partir de nós que tenham definições de variáveis. Um nó k pertencerá a um grafo(i) se existir pelo menos um caminho livre de definição do nó i para o nó k c.r.a pelo menos uma das variáveis definidas em i. Associa-se a cada nó k do grafo(i) um conjunto de variáveis denominado $deff(k) \subseteq defg(i)$. Para qualquer variável $v \in deff(k)$, qualquer caminho $\pi = (i, \ldots, k)$ no grafo(i) é livre de definição c.r.a. v. Os arcos (k, l) do grafo(i) originados a partir de um arco primitivo do grafo def são caracterizados como arcos primitivos no grafo(i). Um arco (k, l) caracterizado como arco primitivo no grafo(i) não é necessariamente um arco primitivo no grafo(i) se aplicado o algoritmo REXFLUXDA nesse grafo. Ainda, é importante observar que um nó ou um arco (primitivo) do grafo de programa G pode dar origem a mais do que um nó ou um arco (primitivo) no grafo(i); isto decorre do fato de que é construído um único grafo(i) para todas as variáveis definidas no nó i [MAL88b]. Então um nó $k \in N$ pode dar origem a q imagens $k_1, k_2, \ldots, k_q$ distintas no grafo(i). Denota-se por $f_{\pi_q}(k)$ a *q-ésima* imagem de k, que ocorre no caminho π_q do grafo(i); se existirem duas ou mais imagens de k em um mesmo caminho π_q, denotam-se por $f_{1\pi_q}(k)$ e por $f_{2\pi_q}(k)$ a primeira e a segunda imagens de k no caminho π_q, respectivamente. É importante salientar que, considerando-se dois caminhos distintos π_1 e π_2, para quaisquer duas imagens distintas no grafo(i), $f_{i\pi_1}(n)$ e $f_{i\pi_2}(n)$, de um nó $n \in N$, $deff(f_{i\pi_1}(n)) \neq deff(f_{i\pi_2}(n))$, $i = 1, 2$.

Adicionalmente, introduz-se o conceito de arco primitivo p_ϕ para garantir-se que qualquer caminho do grafo(i) contenha pelo menos um arco primitivo. Um arco (k, l) do grafo(i) é transformado em arco primitivo p_ϕ se existe um caminho $\pi_q = (i, \ldots, k, l)$ e: **i)** $deff(l) = \phi$ no caminho π_q, ou seja, todas as variáveis definidas no nó i foram redefinidas no caminho π_q ($deff(f_{\pi_q}(l)) = \phi$); ou **ii)** $l = f_{2\pi_q}(n)$, $n \in N$, ou seja, l é a segunda imagem de $n \in N$ no caminho π_q do grafo(i).

Dado um grafo(i) com os seus respectivos arcos primitivos caracterizados conforme descrito acima, a automatização dos critérios Potenciais Usos é realizada com pequenas variações dos descritores dos caminhos e associações requeridos por estes critérios.

Seja um grafo $G = (N, A, s)$ e um grafo(i); define-se $N_i = \{n \in N \mid$ existe caminho π livre de definição c.r.a uma variável $v \in defg(i)$ até n, ou seja, existe $f_\pi(n)$ no $grafo(i)\}$; $N_r = \{k \in grafo(i) \mid k = f_{2\pi}(n), n \in N$, onde π é um potencial-du-caminho do grafo de programa $G\}$; e $N_\phi = \{k \in grafo(i) \mid deff(k) = \phi\}$.

Introduz-se a notação $[i,(j,k),\{v_1,\ldots,v_n\}]$ para representar o conjunto de associações $[i,(j,k),v_1],\ldots,[i,(j,k),v_n]$; ou seja, $[i,(j,k),\{v_1,\ldots,v_n\}]$ indica que existe, no grafo(i), pelo menos um caminho livre de definição c.r.a $v_1,\ldots,v_n$ do nó i ao arco (j,k). Observe-se que podem existir, no grafo(i), outros caminhos livres de definição c.r.a algumas das variáveis $v_1,\ldots,v_n$, mas que não sejam, simultaneamente, livres de definição para todas as variáveis $v_1,\ldots,v_n$. Descreve-se a seguir como caracterizar e descrever os caminhos e associações para alguns dos critérios Potenciais Usos: todos-potenciais-du-caminhos, todos-potenciais-usos e todos-potenciais-usos/du. Esta descrição deve ser feita somente em termos de nós $n \in N$, pois os caminhos efetivamente executados são constituídos de sequências de nós $n \in N$.

Critério todos-potenciais-du-caminhos

Um conjunto de casos de teste T satisfaz o critério todos-potenciais-du-caminhos se levar à execução de todos os caminhos dos grafo(i). Cada um destes caminhos é identificado por uma sequência de arcos primitivos. A partir destas sequências, geram-se os descritores (expressões regulares) desses caminhos, que constituem a base para a construção dos aceitadores utilizados na avaliação da adequação de um dado conjunto T de casos de teste.

Seja um caminho π do grafo(i) constituído pelos arcos primitivos

$$p_1 = (k_1,l_1), p_2 = (k_2,l_2),\ldots,p_n = (k_n,l_n)$$

Se $k_1 \neq i$, o descritor desse caminho é

$$desc_\pi = N^* i N^*_{lf} k_1 l_1 N^*_{lf} k_2 l_2 N^*_{lf} \ldots N^*_{lf} k_n l_n$$

Se $k_1 = i$, então o descritor é

$$desc_\pi = N^* k_1 l_1 N^*_{lf} k_2 l_2 N^*_{lf} \ldots N^*_{lf} k_n l_n$$

Esses descritores em termos de expressões regulares são, na realidade, correspondentes a autômatos finitos. Os autômatos correspondentes aos descritores dos caminhos requeridos pelo critério todos-potenciais-du-caminhos são ilustrados na Figura 4.16a.

O conjunto N_{lf} é igual ao conjunto $N_i/1$, onde $N_i/1$ indica que os elementos de N_{lf} podem ocorrer uma única vez; isto quer dizer que este é um conjunto dinâmico, pois, deve ser atualizado sempre que ocorrer um de seus elementos no caminho avaliado. Esta condição deve ser monitorada dinamicamente na implementação dos aceitadores (ver módulo *Avaliador*, Capítulo 5).

Em geral, a execução de todos os ramos do grafo(i) satisfaz este critério; no entanto, esta condição apesar de ser suficiente não é, em geral, necessária; ou seja, há situações em que, mesmo não tendo sido executados todos os ramos do grafo(i), este critério poderia ter sido satisfeito. Esta situação deve ser considerada principalmente na avaliação de um dado conjunto T de casos de teste e é decorrente de ter-se construído um único grafo(i) para cada no $i \mid defg(i) \neq \phi$. Por exemplo, considere um arco primitivo $p = (j, l)$; suponha que p tenha gerado imagens distintas no grafo(i). Sejam $f_{i\pi_1}(j)$ e $f_{i\pi_2}(j)$ as imagens de j no grafo(i); se $deff(f_{i\pi_2}(j)) \neq \phi$ e $deff(f_{i\pi_1}(j)) \neq \phi$, as associações $a_1 = [i, (j, l), deff(f_{i\pi_1}(j))]$ e $a_2 = [i, (j, l), deff(f_{i\pi_2}(j))]$ seriam então requeridas. Se $deff(f_{i\pi_1}(j)) \subset deff(f_{i\pi_2}(j))$ um caminho que exercitasse a associação a_2, também exercitaria a associação a_1 e, portanto, o arco $(f_{i\pi_1}(j), f_{i\pi_1}(l))$ do grafo(i) não seria necessariamente executado.

Propõe-se, então, para cada arco primitivo $p = (j, l) \in A$, que deu origem a q arcos primitivos no grafo(i), a criação de um descritor para cada variável $v \in defg(i) \cap (\bigcup_m deff(f_{\pi_m}(j)) - \bigcap_m deff(f_{\pi_m}(j)))$, $m = 1, 2, \ldots, q$. Para cada grafo(i) os primitivos p_ϕ são tratados da mesma forma que os primitivos obtidos pela aplicação do algoritmo REHXFLUXDA.

Seja o conjunto $N_{nv} = N_i - (N_i \cap N_v)$, onde o conjunto N_v é constituído pelos nós que têm definição da variável v, ou seja, $N_v = \{n \in N \mid v \in defg(n)\}$.

Se $j \neq i$, os descritores são definidos por

$$N^* i N^*_{nv} j [N^*_{nv} j]^* l.$$

Se $j = i$, os descritores são definidos por

$$N^* i [N^*_{nv} i]^* l,$$

ou, simplesmente, por

$$N^* i l.$$

Isso equivale a requerer, para cada arco primitivo $p \in A$, uma associação $a_n = [i, (j, l), v]$ para cada variável $v \in defg(i) \cap (\bigcup_m deff(f_{\pi_m}(j)) - \bigcap_m deff(f_{\pi_m}(j)))$, $m = 1, 2, \ldots, q$.

Observe-se que a exemplo do critério anterior, *a cada descritor* existe associado *um conjunto de nós* N_{nv} ; no entanto, este conjunto é estático, ou seja, ele permanece inalterado ao longo de todo o processo de avaliação, pois as associações requeridas por esse critério podem ser exercitadas por caminhos que contenham laços.

Esses descritores em termos de expressões regulares são, na realidade, correspondentes a autômatos finitos. Os autômatos correspondentes aos descritores dos caminhos requeridos pelo critério todo-potenciais-usos são ilustrados na Figura 4.16b.

Dessa forma, no pior caso, para um grafo(i) ter-se-ia no máximo $n_v * n_p$ descritores para cada grafo(i), onde n_v indica o número de variáveis definidas em i e n_p o número de arcos primitivos que deram origem a arcos primitivos do grafo(i) mais o número de arcos p_ϕ.

Para a automatização destes princípios e conceitos, requereu-se uma associação $[i,(j,l),deff(f_{\pi_q}(j))]$ para cada ocorrência no grafo(i) de um arco primitivo $p=(j,l) \in A$. A cada associação corresponde um único descritor. Então, se $j \neq i$, o descritor é da forma:

$$N^*iN^*_{nv}j[N^*_{nv}j]^*l.$$

Se $j=i$, os descritores são definidos por

$$N^*i[N^*_{nv}i]^*l$$

ou, simplesmente, por

$$N^*il.$$

O conjunto N_{nv} é redefinido por

$$N_{nv} = N_i - (N_i \cap (\bigcap_{v \in deff(f_{\pi_q}(j))} N_v)),$$

ou seja, o conjunto N_{nv} é obtido a partir do conjunto N_i eliminando-se o conjunto de nós que tenham definição das variáveis que ocorrem no conjunto $deff$ da *q-ésima* imagem do nó j constituinte do arco primitivo (j,l).

Um ponto deve ser salientado nesta última abordagem, que foi a efetivamente implementada. Seja um primitivo $p=(j,l)$ com q ocorrências no grafo(i). Note-se que a execução de uma associação $a_{\pi_w}, 1 \leq w \leq q$, implica a execução de qualquer associação $a_{\pi_m} \mid deff(f_{i\pi_m}(j)) \subset deff(f_{i\pi_w}(j))$. Suponha, agora, que exista um número $n < q$ de imagens de $p \mid \bigcup_m deff(f_{i\pi_m}(j)) \supseteq deff(f_{i\pi_w}(j))$, $m=1,2,\ldots,n$. A execução das associações

$$a_{\pi_m} = [i,(j,l),deff(f_{i\pi_m}(j))],$$

implica, teoricamente, também exercitar a associação

$$a_{\pi_w} = [i,(j,l),deff(f_{\pi_w}(j))],$$

pois, são executados caminhos livres de definição para cada variável $v \in deff(f_{i\pi_w}(j))$; no entanto, esta associação continuaria ainda a ser requerida. Note-se que esta situação é bastante particular por requerer características específicas tanto do *grafo def* como do conjunto de casos de teste. Este aspecto é contornável procurando-se sempre exercitar, entre as associações requeridas, aquelas com maior número de variáveis.

Critério todos-potenciais-usos/du

A distinção básica entre este critério e o critério todos-potenciais-usos é que as associações requeridas devem ser executadas por potenciais-du-caminhos; portanto, os descritores das associações devem incorporar esta restrição, a exemplo do critério todos-potenciais-du-caminhos.

A forma geral dos descritores é fundamentalmente a mesma dos critérios todos-potenciais-usos, porém, restringindo-se a aceitação de caminhos com laços. Seja um arco primitivo $p = (j, l) \in G$ com q ocorrências no grafo(i). Requer-se uma associação $[i, (j, l), deff(f_{\pi_q}(j))]$ para cada ocorrência no grafo(i) de um arco primitivo $p = (j, l) \in A$. A cada associação corresponde um único descritor. Então, se $j \neq i$, o descritor é da forma:

$$N^*iN^*_{nvlf}jl.$$

Se $j = i$, os descritores são definidos por

$$N^*il$$

e o conjunto N_{nvlf} por

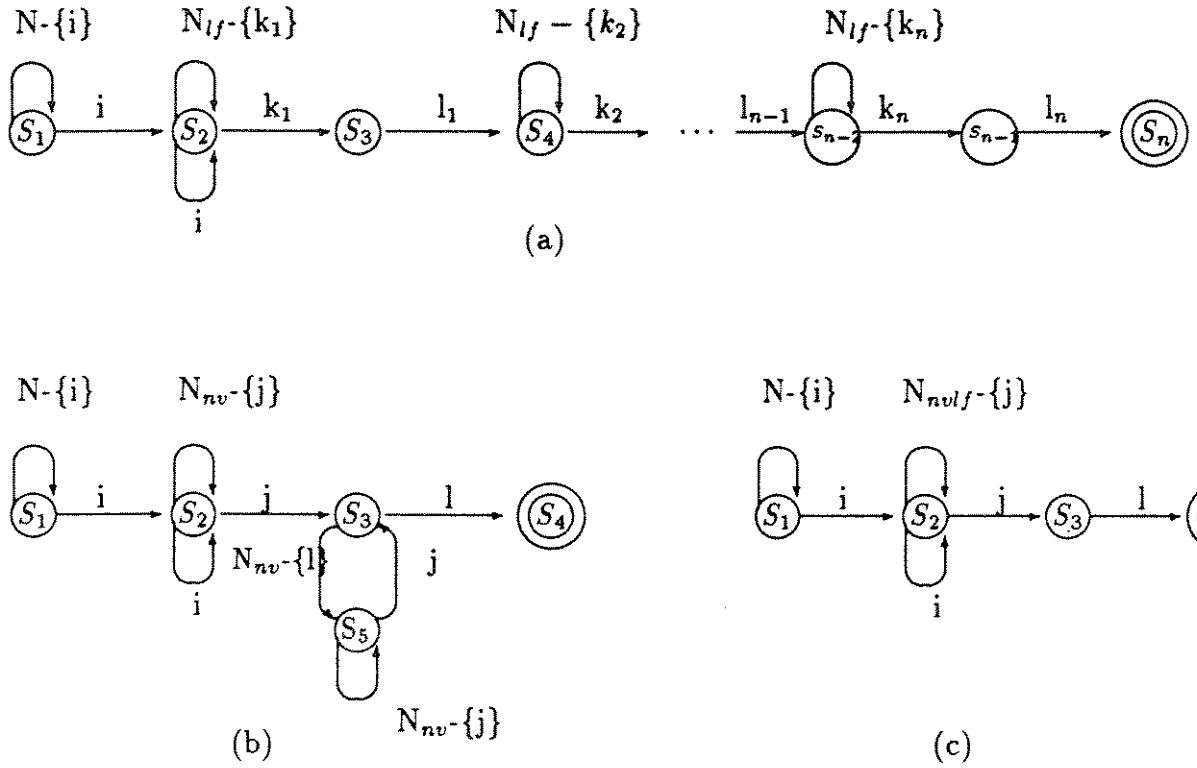
$$N_{nvlf} = N_i - (N_i \cap (\bigcap_{v \in deff(f_{\pi_q}(j))} N_v)),$$

ou seja, o conjunto N_{nvlf} é igual ao conjunto N_{nv}. No entanto, da mesma forma que para o critério todos potenciais-du-caminhos, o conjunto N_{nvlf} é um conjunto dinâmico, isto é, os elementos de N_{nvlf} podem ocorrer uma única vez; ele é atualizado, no processo de avaliação, sempre que ocorrer um elemento pertencente a este conjunto no caminho executado pelo caso de teste. Esta condição deve ser monitorada dinamicamente na implementação dos aceitadores (ver módulo *Avaliador*, Capítulo 5).

Os autômatos correspondentes aos descritores dos caminhos requeridos pelo critério todos-potenciais-usos são ilustrados na Figura 4.16c.

No contexto da POKE-TOOL duas medidas são fornecidas em decorrência da análise de cobertura. Uma delas fornece um percentual de associações exercitadas em relação às requeridas. Por exemplo, para o critério todos-potenciais-du-caminhos é fornecida uma porcentagem dos caminhos executados com relação aos caminhos requeridos. A outra medida fornece uma média dos percentuais de cobertura dos elementos requeridos em relação a cada grafo(i).

Mais especificamente, seja $DEF = \{i \in N \mid defg(i) \neq \phi\}$, *card(U)* uma função que retorna a cardinalidade do conjunto U, *ASSOCEX(i)* o número de associações exigidas



Obs.: Em qualquer estado, a ocorrência do nó i leva o automato para o estado
da mesma forma, a ocorrência de um nó ($\neq i$) não especificado explicitame
te no automato, provoca uma transição para o estado S_1.

Figura 4.16: Automatos Correspondentes aos Descritores dos Critérios Potencias Usos.

por um grafo(i) executadas, *ASSOCTOT(i)* o número total de associações exigidas por um grafo(i), *ASSOCEX* o número de associações executadas e *ASSOCTOT* o número total de associações requeridas. A primeira medida, chamada cobertura total (*CT*), é dada por

$$CT = \frac{ASSOCEX}{ASSOCTOT}.$$

A segunda medida, chamada Cobertura de Fluxo de Dados (*CFD*), é dada por

$$CFD = \frac{\sum_{i \in DEF} \frac{ASSOCEX(i)}{ASSOCTOT(i)}}{card(DEF)}.$$

4.5 Considerações Finais

Visando a fornecer maior precisão e uniformidade à automatização de critérios de teste, principalmente os baseados em fluxo de dados, foram propostos quatro modelos teóricos que abrangem tópicos comuns à automatização de qualquer critério de teste estrutural. Considerando também que um dos objetivos de continuidade deste trabalho é incorporar à POKE-TOOL outros critérios de teste, viabilizando-se estudos comparativos entre os diversos critérios, essa uniformidade de implementação torna-se muito importante.

A principal contribuição reside no modelo de descrição dos elementos requeridos. Considerou-se o uso do conceito de arco primitivo no contexto de fluxo de dados e propôs-se uma variante do algoritmo introduzido por Chusho; esse algoritmo modificado foi denominado *Algoritmo de Redução de Herdeiros de Fluxo de Dados*. Nesse sentido introduziu-se o conceito de *arco primitivo de fluxo de dados*. A partir desses conceitos e do grafo def, foi introduzido o conceito de *grafo(i)*, estabelecendo-se uma base de descrição dos elementos requeridos.

Os outros três modelos — de fluxo de controle, de instrumentação, e de fluxo de dados são também fundamentais para uniformidade e consistência nos estudos de comparação entre esses critérios.

Portanto, espera-se ter estabelecido um núcleo básico para a implementação e suporte automatizado à aplicação dessa classe de critérios. Exemplificando, a automatização do critério *todos ramos* é praticamente imediata, bastando para tanto a criação de um autômato do tipo N^*ij para cada arco primitivo do grafo de fluxo de controle G. Os critérios *estendidos a ciclo* são facilmente automatizados com descritores semelhantes ao do critério *todos-potenciais-usos*; de fato, este último é a extensão a ciclo do critério *todos-potenciais-usos/du*. A Família de Critérios de Fluxo de Dados, por sua vez, requer apenas que na análise estática da unidade sejam identificados os usos das variáveis, pois o *grafo def-uso* é na realidade um sub-grafo do *grafo def*. Todos os demais critérios de fluxo de dados exigem essa extensão na análise estática da unidade em teste, uma vez que é necessário que ocorra um uso explícito para que uma associação seja estabelecida e, consequentemente, seja requerida por esses outros critérios.

Capítulo 5

POKE-TOOL — Arquitetura da Ferramenta de Suporte à Aplicação dos Critérios Potenciais Usos ao Teste Estrutural de Programas

Neste capítulo, apresentam-se a arquitetura e os principais aspectos funcionais e de implementação de uma ferramenta de teste - denominada POKE-TOOL (**PO**tential Uses **CRI**teria **TOOL** for program testing) - que apóia a utilização dos *Critérios Potenciais Usos Básicos*(PU) [MAL88a, MAL88b, MAL89]. Os critérios Potenciais Usos são critérios de teste estrutural, baseados na análise de fluxo de dados. Relembrando, no contexto de teste de software, métodos e critérios de projeto de casos de teste baseados em análise de fluxo de dados requerem que as interações que envolvem definições de variáveis de programa e subseqüentes referências afetadas por essas definições sejam testadas; por sua vez, os Critérios Potenciais Usos requerem associações que implicam o exercício de caminhos entre a definição de uma variável e um possível (potencial) uso desta variável; adicionalmente às propriedades dos Critérios Potenciais Usos, discutidas no Capítulo 3, o conceito *potencial uso* facilita a implementação desses critérios, uma vez que na análise estática da unidade em teste não é necessária a determinação da ocorrência de usos de variáveis, por exemplo.

Analogamente aos outros critérios, os critérios Potenciais Usos necessitam de uma ferramenta para sua aplicação efetiva, pois a aplicação dos critérios baseados em análise de fluxo de dados sem o apoio de uma ferramenta automatizada é limitada a programas muito simples [KOR85]. A importância de tais ferramentas tem sido discutida por muito outros autores [NTA88, FRA85, WEY84, PRI87, LAS83, WHI85] e algum esforço para implementar tal classe de ferramenta tem sido feito [FRA85, KOR85, WHI85]. Ainda, à medida que os sistemas de software cresceram em tamanho e complexidade, o esforço requerido para testar esses sistemas tem crescido muito além das expectativas, implicando altos custos e produtos com baixa confiabilidade. Essa situação

ensejou o desenvolvimento de ferramentas automáticas para auxiliar na produção de testes efetivos e na análise dos resultados dos testes. Chaim [CHA91d] descreve essas e outras ferramentas de apoio ao teste estrutural de programas.

As ferramentas de teste estrutural de software fazem, em quase sua totalidade, análise de cobertura, segundo algum critério de teste selecionado, de um conjunto de casos de teste. Algumas delas oferecem algum suporte para a obtenção dos dados de entrada necessários para satisfazer um particular requisito de teste; em geral, esse suporte é baseado em execução simbólica do programa em teste. Apesar de não auxiliarem diretamente na determinação das entradas de um programa, necessárias para a execução de caminhos específicos, a maioria das ferramentas de teste apresentam, ao usuário, quais os requisitos de teste exigidos para que os critérios sejam satisfeitos. Assim, de certa maneira, orientam e auxiliam os usuários na elaboração dos casos de teste.

As ferramentas comerciais, descritas em [CHA91d], são, em geral, sofisticadas em termos de relatórios fornecidos e interface com o usuário; outra característica de sofisticação é que elas permitem ao usuário fazer a análise de cobertura de várias unidades simultaneamente. Se por um lado essa característica possibilita testar várias unidades concomitantemente, por outro, impede que certos caminhos (que são executados somente através do teste isolado da unidade) sejam testados. O teste isolado da unidade implica a construção de unidades "drivers" e "stubs"; o ideal seria que as ferramentas fornecessem para o usuário um arcabouço dessas unidades; entretanto, nenhuma dessas ferramentas de teste comerciais têm essa capacidade. Essas ferramentas são todas voltadas para uma única linguagem de programação, apesar de possuírem versões para uma série de linguagens. Essa característica monolinguagem pode ser entendida como uma tentativa de tornar as ferramentas mais eficientes na fase de análise estática do código fonte ou como um estratégia de vendas; uma alternativa é o uso algoritmos configuráveis para a realização da análise estática, que é o enfoque adotado na proposição da POKE-TOOL. A utilização da POKE-TOOL para a aplicação do benchmark (discutido no Capítulo 6) mostrou que esta abordagem não é um gargalo no desempenho das ferramentas.

Um fato importante a ser observado é que não existem ferramentas de teste estrutural de suporte à aplicação dos critérios baseados em fluxo de dados. Todas as iniciativas nesse sentido constituem, presentemente, protótipos desenvolvidos nas universidades [HER76, KOR85, WHI85, FRA87]. As ferramentas de teste orientadas para o teste com critérios baseados em análise de fluxo de dados implementam, com exceção de PROTESTE [PRI90], somente uma família de critérios. Essas ferramentas diferem na abordagem com que são determinados os requisitos de teste e nas características encontradas em cada uma quanto à interface com o usuário, facilidades para determinação dos dados de entrada, facilidades para tratar várias linguagens e gerenciamento da sessão de trabalho. A arquitetura da POKE-TOOL é fortemente inspirada na ferramenta de teste ASSET [FRA87].

A POKE-TOOL é uma ferramenta que apóia a aplicação dos critérios Potenciais

Usos para o teste de unidades; as unidades são entendidas aqui como procedimentos de uma linguagem procedimental. Os critérios Potenciais Usos podem ser utilizados, tanto para auxiliar na seleção de casos de teste como para avaliar a adequação de um determinado conjunto de casos de teste em relação a estes critérios. A POKE-TOOL é uma ferramenta flexível, isto é, não atrelada a uma linguagem de programação específica, permitindo que o usuário a configure para a linguagem de programação de seu interesse.

Com o desenvolvimento da ferramenta POKE-TOOL pretende-se, além de auxiliar o uso prático dos critérios Potenciais Usos, viabilizar a realização de comparações entre estes critérios e os demais critérios de teste estrutural, bem como avaliar a adequação destes critérios a classes de erros; neste sentido, outros critérios deverão ser incorporados à POKE-TOOL, procurando-se utilizar os mesmos modelos discutidos no Capítulo 4.

Na Seção 5.1 é discutida a Arquitetura da Ferramenta POKE-TOOL que suporta a aplicação dos Critérios Potenciais Usos; nas Seções 5.2 e 5.3 é apresentada uma síntese dos aspectos do projeto e da implementação e das etapas para configuração da POKE-TOOL, respectivamente, extraída de [CHA91a, CHA91b].

5.1 Arquitetura da Ferramenta POKE-TOOL

Segundo Deutsch [DEU82], existem 5 funções básicas que devem ser realizadas por uma ferramenta de suporte às atividades de testes de programas:

i) análise do código fonte e criação de uma base de dados;

ii) geração de relatórios baseada em análise estática do código fonte, que indique problemas potenciais ou existentes e identifique a estrutura de dados e de controle do software;

iii) instrumentação do código fonte permitindo a coleta de dados relativos à execução de casos de teste;

iv) análise dos resultados dos testes e geração de relatórios; e

v) geração de relatórios de apoio ao teste para auxiliar na organização das atividades de teste e para derivar conjuntos de entrada (casos de teste) para testes específicos.

Quanto à ferramenta POKE-TOOL, pretende-se apoiar todas essas funções da atividade de teste, sendo que, inicialmente, foi implementada uma primeira versão que apóia o teste de programas escritos na linguagem C e implementa, pelo menos parcialmente, todas as funções citadas acima.

Semelhantemente à ferramenta de teste implementada por Frankl e Weyuker – ASSET [FRA85] –, a POKE-TOOL [MAL89][1], cuja arquitetura geral está ilustrada na Figura 5.1, tem como entrada o programa a ser testado, a seleção do critério a ser utilizado e um conjunto de casos de teste. Na hipótese do conjunto de casos de teste fornecido ser vazio, a POKE-TOOL fornecerá um conjunto de caminhos necessários para satisfazer o critério selecionado, orientando desta forma a própria seleção de casos de teste. Se o conjunto de casos de teste fornecido não for vazio, será produzida uma relação de caminhos requeridos pelo critério mas ainda não executados. Pretende-se dar suporte para que esta ferramenta aceite programas implementados em linguagens de programação distintas, tais como PASCAL, C, COBOL, etc. Este fato exige um pré-processamento do programa fonte de forma que as particularidades de cada uma das linguagens sejam tratadas uniformemente.

Como pode ser extraído da Figura 5.1, a POKE-TOOL é constituída basicamente de nove funções descritas abaixo:

GRAFO DE FLUXO DE CONTROLE: Esta função produz o grafo de fluxo de controle do programa fonte a ser testado. Para a produção do grafo de fluxo de controle para várias linguagens utiliza-se uma ferramenta [CAR91] que, a partir do código fonte e com base em uma tabela descritora da sintaxe da linguagem de implementação, gera uma linguagem intermediária que é utilizada para geração do fluxo de controle do programa. Note-se que a incorporação de novas linguagens far-se-á unicamente com a inserção das tabelas descritoras que permitem à ferramenta produzir a linguagem intermediária a partir do código fonte. A correspondência semântica entre os comandos da linguagem fonte e os comandos da linguagem intermediária faz parte do processo de configuração e é de responsabilidade do usuário configurador.

CÁLCULO DOS ARCOS PRIMITIVOS: calcula os arcos primitivos (definidos na Seção 4.4.1)do grafo de fluxo de controle. O conjunto de arcos primitivos servirá de base para a instrumentação do programa fonte e para a construção dos autômatos utilizados na avaliação da adequação de conjuntos de casos de teste. O algoritmo utilizado para determinação dos arcos primitivos, descrito neste relatório, é uma variante do algoritmo de Chusho [CHU87]. Esta variante consiste numa adequação desse algoritmo para a utilização do conceito de arcos primitivos no contexto de testes estruturais baseados em análise de fluxo de dados.

EXTENSÃO DO GRAFO DE FLUXO DE CONTROLE: associa a cada nó i do grafo de fluxo de controle o conjunto de variáveis definidas no nó i, produzindo um grafo denominado grafo def, de acordo com um modelo de dados pré-estabelecido.

INTERFACE GRÁFICA: apresenta os grafos de fluxo de controle para o usuário.

[1] O nome da ferramenta foi mudado de FERCRIPU para POKE-TOOL.

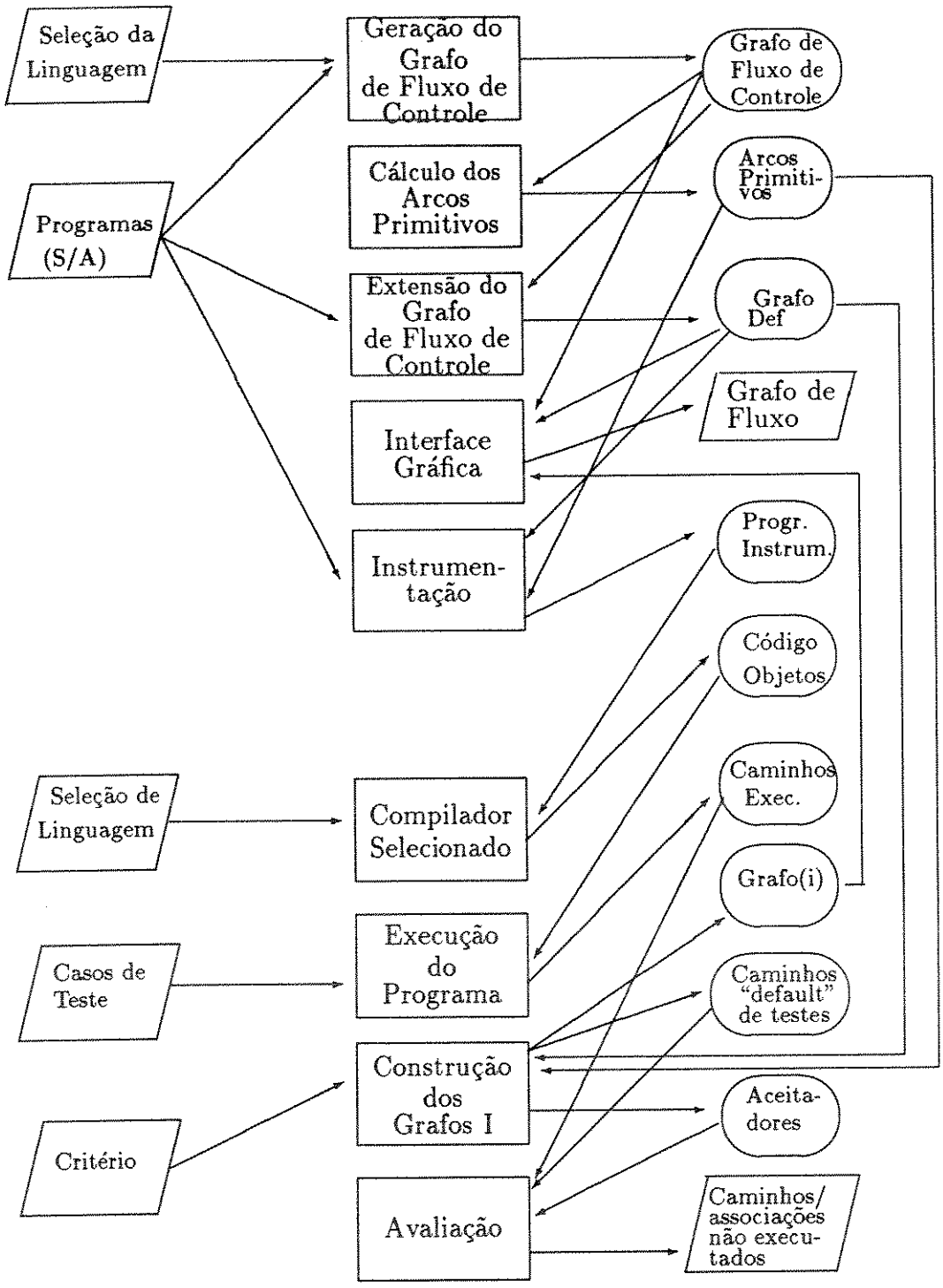


Figura 5.1: Arquitetura da Ferramenta de Teste Estrutural POKE-TOOL.

INSTRUMENTAÇÃO: insere comandos de escrita (pontas de prova) no programa fonte para cada um dos nós da unidade em teste, gerando uma nova versão do programa – versão instrumentada – que produz um "trace" da execução dos casos de teste.

COMPILADOR SELECIONADO: consiste de um compilador da linguagem fonte na qual o programa em teste foi implementado.

EXECUÇÃO DO PROGRAMA: controla a execução do programa em teste – unidade em teste – produzindo um conjunto dos caminhos executados pelos casos de teste fornecidos. Adicionalmente, produz o registro de pares <entrada, saída> associados aos respectivos caminhos executados. Os caminhos executados são descritos através de seqüências de números dos nós do grafo de fluxo de controle do programa fonte.

CONSTRUÇÃO DOS GRAFO(i): com base no grafo def e no conjunto de arcos primitivos esta função constrói os grafo(i) [MAL88b]. Adicionalmente, fornece um conjunto de caminhos e associações requeridos para satisfazer os critérios Potenciais Usos. Ainda são fornecidos os descritores de caminhos e associações, em termos dos arcos primitivos, a serem utilizados na avaliação de um conjunto de casos de teste qualquer.

AVALIAÇÃO: esta função verifica se o conjunto de caminhos ou associações executados satisfaz o critério selecionado. Em caso negativo, produz uma relação de caminhos (associações) requeridos pelo critério e não executados e uma medida percentual da cobertura provida pelo conjunto de casos de teste, ou seja, uma relação entre os caminhos (associações) executados e o número de caminhos (associações) requeridos.

De forma resumida, pode-se dizer que a partir do programa fonte, a POKE-TOOL determina o grafo de fluxo de controle. A seguir, este grafo é estendido incorporando-se informações de fluxo de dados obtendo-se o grafo def; o conjunto de arcos primitivos é calculado, utilizando-se o algoritmo REHFLUXDA, obtendo-se o grafo com redução de herdeiros para fluxo de dados. Estes arcos primitivos são utilizados para construir descritores (expressões regulares) dos caminhos (associações) requeridos. A instrumentação auxilia na determinação dos caminhos efetivamente executados pelo conjunto de casos de teste fornecido. Os descritores são utilizados para verificar se o critério selecionado foi satisfeito; esta verificação dá-se pela implementação de aceitadores de expressões regulares. A partir do grafo def e do conjunto de arcos primitivos constróem-se os grafo(i), caracterizando-se os arcos primitivos em cada um desses grafo(i). Em seguida, os descritores dos caminhos (associações) requeridos são elaborados. Na hipótese da POKE-TOOL ser utilizada na avaliação da adequação de um conjunto de casos de teste, são determinados os caminhos (associações) efetivamente executados e verifica-se se os aceitadores correspondentes aos descritores dos caminhos (associações) requeridos estão no estado final (o critério foi satisfeito); caso contrário, é fornecida ao usuário uma lista de caminhos requeridos pelo critério e não executados

pelo conjunto de casos de teste. No caso da POKE-TOOL ser utilizada para auxiliar na geração de casos de teste, o conjunto "default" de caminhos requeridos pelo critério (obtidos durante a construção dos grafo(i)) é fornecido.

Entre os caminhos requeridos pode existir um caminho não executável; a ferramenta POKE-TOOL, na sua primeira versão, não dá nenhum suporte para determinação da executabilidade de um determinado caminho – o que é uma questão indecidível –; métodos heurísticos foram desenvolvidos para tratar esse problema [FRA85, FRA88]. Estudos para incorporar na POKE-TOOL facilidades para tratamento de caminhos não executáveis e estudos de comparação entre métricas e heurísticas para previsão e determinação de caminhos não executáveis estão sendo conduzidos [VER91]. Esses estudos contribuirão para a aplicação mais efetiva dos critérios Potenciais Usos e para a definição de estratégias de geração de casos de teste baseadas em fluxos de dados.

Observe-se que a ferramenta POKE-TOOL pode constituir-se também num suporte extremamente útil tanto às atividades de depuração como às de manutenção de programas, pois gera informações usualmente imprescindíveis a essas atividades.

5.2 Aspectos do Projeto e da Implementação da POKE-TOOL

Nesta Seção, para uma melhor compreensão do trabalho como um todo, os principais aspectos do projeto, da implementação e da operação da POKE-TOOL são sintetizados; discussões mais detalhadas destes tópicos são apresentadas em [CHA91a, CHA91b, CHA91c]. A versão atual apóia a aplicação dos critérios Potenciais Usos Básicos para o teste de unidades escritas na linguagem de programação C, em ambientes do tipo PC.

A POKE-TOOL foi projetada como uma ferramenta interativa cuja operação é orientada para *sessão de trabalho.* Na sessão de trabalho, o usuário realiza todas as tarefas de teste, a saber: análise estática da unidade; preparação para o teste; submissão de casos de teste; avaliação de casos de teste; e gerenciamento dos resultados de teste. Na POKE-TOOL, a sessão de trabalho é dividida em duas fases: uma *estática* e outra *dinâmica.*

Na fase estática a ferramenta analisa o código fonte, obtém as informações necessárias para a aplicação dos critérios e instrumenta o código fonte, com inserção de pontas de prova (instruções de escrita), que produzem um "trace" do caminho executado, gerando uma nova versão da unidade em teste — *versão instrumentada* —, que viabiliza a posterior avaliação da adequação de um dado conjunto de casos de teste. Terminada essa fase, a POKE-TOOL apresenta, sob solicitação do usuário, entre outras coisas, o conjunto de caminhos requeridos pelo critério *todos potenciais-du-caminhos* e o conjunto de associações requeridas pelo critério *todos potenciais-usos* e *todos potenciais-usos/du.* Com estas informações o usuário pode projetar seus casos de teste a fim de que eles executem os caminhos ou associações exigidos.

A fase dinâmica consiste no processo de execução e avaliação de casos de teste. Porém, antes de executar os casos de teste, é necessário que se gere o programa executável a partir da versão instrumentada da unidade a ser testada. A POKE-TOOL, depois da execução dos casos de teste, realiza a avaliação destes de acordo com um dos três critérios Potenciais Usos. O resultado da avaliação é um conjunto de caminhos ou associações que restam ser executados para satisfazer os critérios e o percentual da cobertura do conjunto de casos de teste. Ainda nesta fase, a ferramenta fornece o conjunto de caminhos ou associacões que foram executados, as entradas e saídas, bem como os caminhos percorridos por cada um dos casos de teste. O processo de execução/avaliação deve continuar até que todos os caminhos ou associações restantes tenham sido satisfeitos, executados ou detectada a sua não executabilidade. Eventualmente, o usuário pode querer interromper a sessão de trabalho sem ter atingido uma das duas situações anteriores. Neste caso, a POKE-TOOL fornece meios para a interrupção da sessão, armazenamento dos dados gerados e do estado da ferramenta até o momento. Posteriormente, pode-se recuperar a sessão de trabalho e recomeçar os testes, ou seja, o teste da unidade pode ser feito incrementalmente.

A POKE-TOOL é composta de vários módulos que se comunicam através de *arquivos*. Esses módulos implementam *funções* ou parte de funções descritas anteriormente na arquitetura da ferramenta. Na Figura 5.2 é apresentado um diagrama contendo os módulos e os diversos produtos gerados. Nessa figura, os *retângulos* representam os módulos, os *losangos* representam as entradas fornecidas pelos usuários à ferramenta e os *círculos* os produtos gerados; as linhas tracejadas representam o *fluxo de controle* e as linhas cheias o *fluxo de informação*.

O módulo **poketool** é o responsável pela comunicação entre a ferramenta e o usuário e pela seqüenciação das atividades de teste através da ativação dos demais módulos.

O módulo **li** é sensitivo à linguagem na qual está escrita a unidade em teste, pois realiza a tradução dessa unidade para uma versão escrita na *linguagem intermediária (LI)*.

O módulo **chanomat** gera o grafo de fluxo de controle (GFC) da unidade — *grafo de programa* — e uma nova versão da unidade em LI onde cada comando está associado ao nó correspondente. Estes dois módulos implementam a função *Grafo de Fluxo de Controle* da POKE-TOOL.

O módulo **pokernel** é o responsável pelo restante da análise estática da unidade em teste, gerando as informações estáticas adicionais, necessárias ao teste dinâmico da unidade. O **pokernel** implementa as seguintes funções da POKE-TOOL: *Cálculo dos Arcos Primitivos*; *Extensão do Grafo de Fluxo de Controle*; *Instrumentação*; *Construção dos Grafo(i)* e *Geração dos Descritores*.

O módulo **gera executável** fornece as condições para a geração do programa executável da versão instrumentada e engloba a função *Compilador Selecionado*.

O módulo **executa caso de teste**, como o próprio nome diz, controla a execução

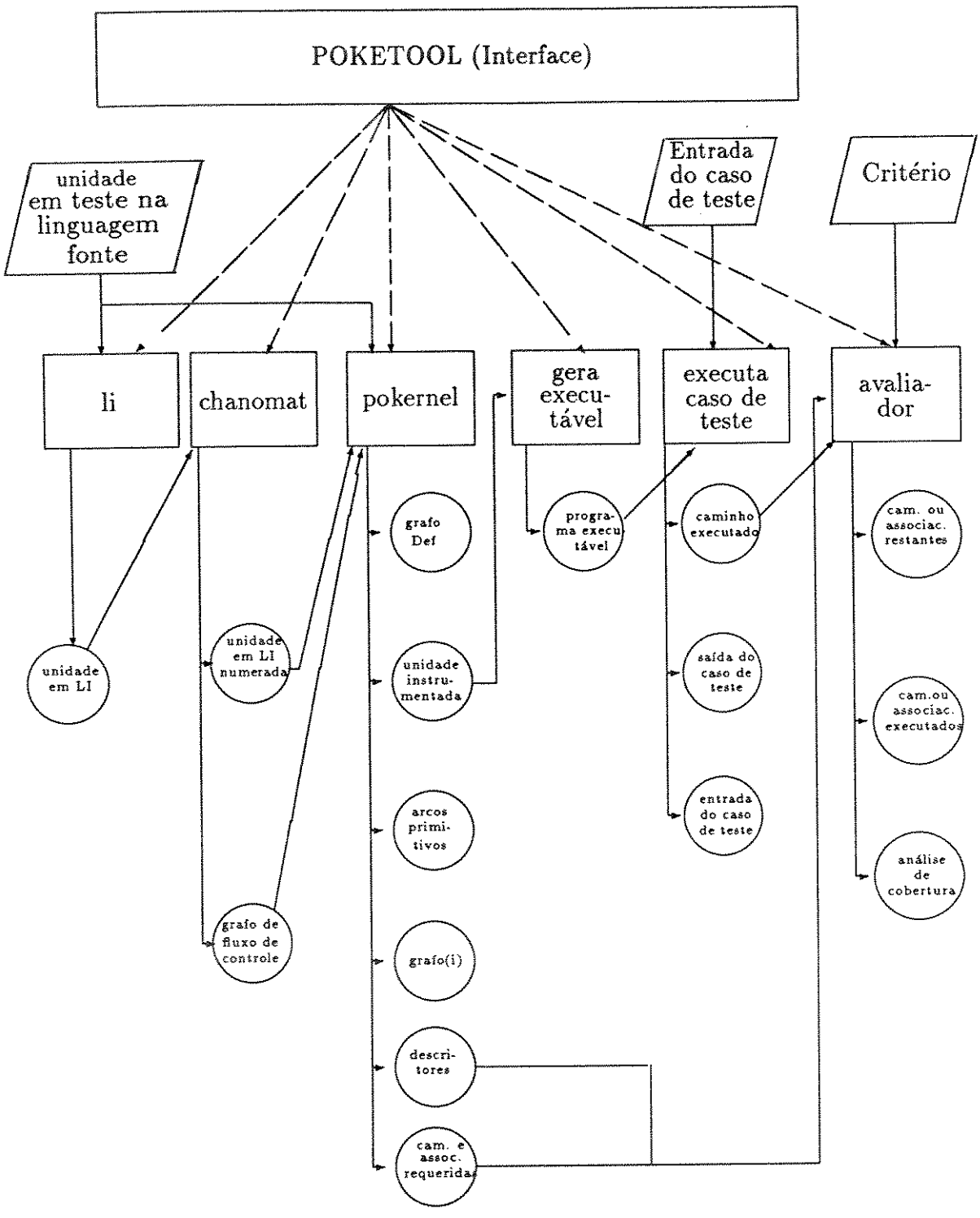


Figura 5.2: Projeto da POKE-TOOL

dos casos de teste salvando as entradas, a saída e o caminho executado para cada caso de teste; implementa a função *Execução do Programa* da POKE-TOOL.

Finalmente, o módulo **avaliador** identifica os caminhos ou associações executados pelos casos de teste e fornece uma análise da cobertura do conjunto de casos de teste fornecido, implementando a função *Avaliação* da POKE-TOOL.

A atual implementação da POKE-TOOL foi desenvolvida para o sistema operacional MS-DOS e foi escrita na linguagem C, sendo que a configuração presente suporta o teste também de unidades (funções) escritas em C. Está em andamento a configuração da POKE-TOOL para as linguagens COBOL e FORTRAN.

Nas seções seguintes são discutidos, de forma um pouco mais detalhada, as funções dos módulos componentes da POKE-TOOL e os principais diretrizes de suas implementações.

5.2.1 Módulo Poketool

O módulo **poketool** é o responsável pela interface da POKE-TOOL com o usuário e pela seqüenciação das atividades de teste através da ativação dos outros módulos da ferramenta.

O enfoque da interface da POKE-TOOL é de orientar a realização das atividades de teste dentro de uma sessão de trabalho de maneira interativa.

O módulo **poketool** em si é um programa executável que realiza a interface com o usuário através de *menus* e recebendo entradas via teclado. Por uma questão de simplicidade, na interface não é utilizado nenhum recurso gráfico.

A interface é dividida em telas que representam as diversas atividades de teste. A Figura 5.3 apresenta uma hierarquia das telas da POKE-TOOL. É possível observar, nessa figura, que a própria hierarquia de telas da interface possui uma semelhança muito grande com os módulos constituintes da ferramenta e, por si só, já estabelece a sequenciação das atividades de teste.

A tela INICIAÇÃO prepara o ambiente para o início do processo de teste. Por exemplo, é nessa tela que é perguntado ao usuário se ele deseja iniciar uma nova sessão de trabalho ou recomeçar uma anterior; é também na INICIAÇÃO que é recebido o nome da unidade a ser testada e são ativados os módulos **li**, **chanomat** e **pokernel**[2] para a análise estática da unidade (caso o usuário deseje iniciar uma nova sessão de trabalho).

A tela MENU PRINCIPAL faz a sequenciação das atividades posteriores à análise estática da unidade, ativando as demais telas desde que algumas restrições tenham

[2]A ativação dos módulos **li**, **chanomat** e **pokernel**, que também são programas executáveis, é feita via uma chamada ao sistema operacional.

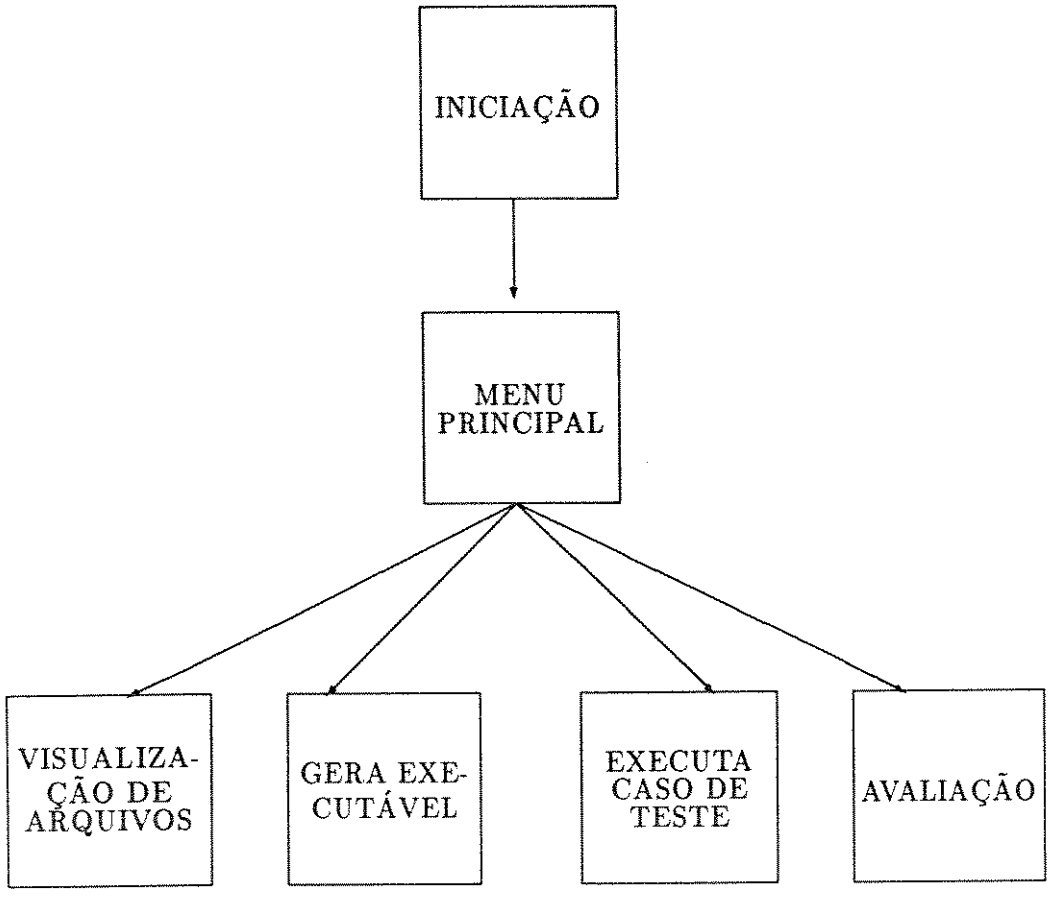


Figura 5.3: Hierarquia das Telas da POKE-TOOL

sido observadas. Assim, o usuário não conseguirá ativar as telas EXECUTA CASO DE TESTE e AVALIAÇÃO se não tiver sido gerada a unidade modificada para teste. A tela MENU PRINCIPAL contém, então, 5 opções; quatro delas causam a seleção das atividades controladas pela POKE-TOOL com a ativação da tela correspondente. A outra – *termina sessão* – termina a sessão de trabalho salvando opcionalmente, em um diretório, os resultados e informações da sessão de teste.

Uma dessas atividades é a VISUALIZAÇÃO DE ARQUIVOS gerados pela ferramenta e desempenhada através da tela de mesmo nome. Essa tela apresenta arquivos gerados tanto na fase estática quanto na fase dinâmica[3]. Tipicamente, os resultados da fase estática apresentados são: o conjunto de arcos primitivos, o grafo def, a unidade modificada para teste, o conjunto de caminhos requeridos pelo critério todos-potenciais-du-caminhos e as associações requeridas pelos critérios todos-potenciais-usos e todos-potenciais-usos/du. Da fase dinâmica são apresentados: as entradas de parâmetros para cada caso de teste (parâmetros aqui se referem aos parâmetros passados na linha de comando para o sistema operacional quando o programa executável, que contém a unidade em teste, é executado), entradas efetuadas via teclado (caso isto ocorra na unidade em teste), as saídas na tela, os caminhos executados, o conjunto de potenciais du-caminhos efetivamente executados, o conjunto de associações efetivamente executadas para o critério todos-potenciais-usos e para o critério todos-potenciais-usos/du. Note-se, entretanto, que algumas restrições devem ser observadas antes de se apresentar alguns desses arquivos. Assim, não são apresentadas as entradas, as saídas ou caminhos executados dos casos de teste se nenhum deles foi executado; não é possível apresentar os conjuntos de caminhos e associações executados se não houve a avaliação de pelo menos um caso de teste. A consistência entre as opções de visualização e o estado atual do processo de teste é verificada pela POKE-TOOL.

A tela GERA EXECUTÁVEL, implementada na interface da POKE-TOOL, na verdade, é a implementação do módulo **gera executável** descrito mais à frente.

Algo semelhante ocorre com a tela EXECUTA CASO DE TESTE e o módulo **executa caso de teste**. Explicações mais detalhadas a respeito dessa tela são apresentadas no módulo **executa caso de teste**.

Na tela AVALIAÇÃO tem-se um menu onde aparecem opções para os três critérios PU. Ao escolher um dos critérios, a interface faz uma chamada ao módulo **avaliador** (que é um programa executável) e, quando termina a chamada, apresenta o arquivo resultado contendo os caminhos ou associações requeridos pelo critério selecionado mas não executados. A interface só torna a chamar o **avaliador** se observar que foi executado mais um caso de teste; caso contrário, é apresentado o resultado obtido da última avaliação.

Para terminar uma sessão o usuário deve retornar ao MENU PRINCIPAL e selecionar a opção *termina a sessão*; a ferramenta vai perguntar se o usuário deseja salvar em

[3]Para apresentar o arquivo na tela é utilizada uma chamada de sistema ao utilitário *more*, presente na versão 3.3 e superiores do sistema operacional MS-DOS.

um diretório os dados gerados na sessão de trabalho até o momento; em caso afirmativo, é criado um diretório com o nome da unidade, se já não existir, e todos os dados gerados pela POKE-TOOL são salvos nesse diretório. Para salvar o estado da ferramenta naquele momento, é criado o arquivo POKELOG.TES que salva as seguintes informações:

(1) nome da unidade em teste;

(2) quantos casos de teste foram executados;

(3) quantos casos de teste foram avaliados para o critério todos-potenciais-du-caminhos;

(4) quantos casos de teste foram avaliados para o critério todos-potenciais-usos;

(5) quantos casos de teste foram avaliados para o critério todos-potenciais-usos/du;

(6) se a unidade aceita parâmetros e

(7) se ela aceita entrada pelo teclado.

No Apêndice C é apresentado um exemplo de utilização da POKE-TOOL com a descrição em "hard copy" da interface da ferramenta.

5.2.2 Módulo Li

Este módulo faz a tradução da unidade em teste para a unidade em teste escrita na linguagem intermediária (LI).

Unidades em LI são a entrada para o módulo **chanomat** que determina o grafo de fluxo de controle (GFC) da unidade.

A LI é uma linguagem com sintaxe e semântica muito simples, composta fundamentalmente com os comandos que alteram o fluxo de execução. A idéia aqui é ter um módulo, no caso o **li**, sensitivo à linguagem da unidade e que gera a unidade em LI, e um outro módulo, no caso **chanomat**, "insensitivo" à linguagem e que calcula o GFC. Assim, o que se deseja é isolar a sensibilidade à linguagem no módulo **li** e deixar o módulo **chanomat** como uma ferramenta de cálculo do GFC.

O módulo **li** é um programa executável que pode ser configurado para fazer a tradução de uma determinada linguagem para a LI. A configuração do módulo **li** deve ser feita por um usuário configurador [CAR91] e envolve a criação de tabelas e rotinas específicas para a tradução da linguagem em questão para a LI. No apêndice C é fornecido um exemplo desta tradução para um dos programas do benchmark.

5.2.3 Módulo Chanomat

Este módulo calcula o grafo de fluxo de controle (GFC), segundo o modelo proposto na seção 4.1, e também produz uma versão da unidade em LI com um campo a mais nos átomos da LI indicando a qual nó pertence o átomo.

O **chanomat** é também um programa executável; sua entrada é o arquivo que contém a versão da unidade em LI e suas saídas são dois arquivos, um contendo uma representação do GFC e outro a nova versão da unidade.

Basicamente, o módulo **chanomat** constitui-se de um analisador sintático descendente para a LI que, durante o processo de reconhecimento do programa LI, vai criando o grafo de fluxo de controle e acrescentando um campo a mais nos átomos da LI. Este campo a mais identifica o nó associado ao átomo. Os novos átomos da LI são definidos pela seguinte produção:

$< atm\_nli > ::= < atomo >< no >< inicio >< comprimento >< linha >$
onde
$< no > ::= NUM.$

O arquivo que contém a descrição do GFC é organizado da seguinte maneira. O primeiro número que aparece indica o número de nós do grafo, os demais números são divididos em listas finalizadas pelo número 0. O primeiro número da lista indica um nó do GFC e os demais números até o 0 são sucessores do primeiro. Note-se que o número 0 é tão somente um finalizador de listas, pois não existe nenhum nó com esse número.

No Apêndice C é apresentado, no arquivo ENTAB.GFC, o grafo de fluxo de controle (GFC) para o programa ENTAB.C do benchmark e a nova versão da unidade em LI, no arquivo ENTAB.NLI.

5.2.4 Módulo Gera Executável

O módulo **gera executável** está implementado na interface da POKE-TOOL, ou seja, no programa executável **poketool**. Consiste unicamente numa tela da interface que explica ao usuário como compilar a versão da unidade gerada pela POKE-TOOL com os demais arquivos que compõem o programa a ser executado. Em seguida, é fornecido ao usuário um "shell" do sistema operacional onde ele pode utilizar o seu compilador preferido para recompilar o programa com a unidade modificada para teste e torná-lo apto aos testes. Findo esse processo, o usuário retorna à POKE-TOOL e começa a executar os casos de teste.

Na versão atual da POKE-TOOL não ocorre a geração automática de programas "drivers" e "stubs"; portanto, o usuário deve gerar esses programas. Também não é fornecida ao usuário re-compilação automática da unidade em teste e das demais unidades que compõem o programa executável. Pretende-se incluir essas características em versões futuras da ferramenta.

5.2.5 Módulo Executa Caso de Teste

Este módulo é implementado parte na interface e parte através de um programa executável chamado **ex_prog**.

No módulo **poketool**, através da tela EXECUTA CASO DE TESTE, ocorre a entrada de parâmetros para a chamada do programa executável que contém a unidade em teste modificada; isto se o programa aceitar parâmetros. Na primeira vez que o usuário executa um caso, a tela EXECUTA CASO DE TESTE pergunta ao usuário se o programa aceita parâmetros de entrada e se ele aceita entrada pelo teclado. A ferramenta salva essas informações e, nas próximas execuções, o usuário já entra com os parâmetros do programa, se necessário, sendo que essas entradas são salvas em arquivo. Em seguida, a interface faz uma chamada especial de sistema para o programa **ex_prog**que tem o efeito de terminar a execução do módulo **poketool** e provocar a execução do **ex_prog**. Os parâmetros passados para a execução do **ex_prog** são os parâmetros fornecidos pelo usuário mais as informações para re-direcionar a entrada via teclado (se houver esse tipo de entrada) e a saída na tela. As informações de re-direcionamento servem para que, posteriormente, na VISUALIZAÇÃO DE ARQUIVOS, o usuário possa obter a entrada via teclado e a saída na tela para cada caso de teste.

A função do programa **ex_prog** é simplesmente fazer uma chamada de sistema comum para a execução do programa executável, com os devidos parâmetros e informações de re-direcionamento. Terminada a execução do programa, o **ex_prog** faz uma chamada especial de sistema que causa o fim do **ex_prog** e provoca a execução do **poketool** e, consequentemente, o retorno da execução da *interface*.

Foi adotada essa solução para a execução de casos de teste porque, se fosse feita uma chamada normal de sistema para o programa em teste, dentro do módulo **poketool**, poderia acontecer de, durante a execução do programa em teste, ocorrer uma invasão de memória que poderia atingir o código do módulo **poketool** e, quando fosse retornada a execução para o **poketool**, não estaria garantida a perfeita execução da interface. Com a solução adotada, está garantido que se ocorrer uma invasão de memória o defeito ocorrerá durante a execução do caso de teste e, dessa maneira, estará se detectando um defeito da unidade em teste. Cabe ressaltar ainda que essa solução foi necessária porque foi utilizado o sistema operacional MS-DOS que não dá garantias durante as chamadas de sistema; isto não ocorreria em sistemas operacionais mais sofisticados.

5.2.6 Módulo Pokernel

Este módulo é o responsável por parte da análise estática do código fonte da unidade. Esta análise é fundamental para a aplicação dos critérios Potenciais Usos, pois estes critérios são critérios caixa-branca, ou seja, derivam seus *requisitos* para o teste a partir da estrutura da unidade.

Obviamente, uma análise estática inicial já foi feita quando o módulo **chanomat** determinou o grafo de fluxo de controle e a unidade em LI numerada. Entretanto, é o **pokernel** que gera:

(1) o conjunto de arcos primitivos (arquivo ARCPRIM.TES),

(2) o grafo def (arquivo GRAFODEF.TES),

(3) a unidade instrumentada (arquivo TESTEPROG.C),

(4) o conjunto de potenciais du-caminhos requeridos pelo critério todos potenciais-du-caminhos (arquivo PDUPATHS.TES),

(5) as associações requeridas pelos critérios todos potenciais-usos e todos potenciais-usos/du (arquivo PUASSOC.TES),

(6) os descritores para os critérios Potenciais Usos (arquivos DES_PDU.TES, DES_PU.TES e DES_PUDU.TES).

Note-se que o módulo **pokernel** tem como entrada os arquivo UNIDADE.C (na versão atual da POKE-TOOL), UNIDADE.GFC e UNIDADE.NLI onde *UNIDADE* é o nome da unidade em teste e os arquivos mencionados contêm, respectivamente, a unidade propriamente dita, o grafo de fluxo de controle e a versão da unidade em LI numerada.

O primeiro passo desse módulo é calcular os arcos primitivos do GFC. O conceito de arco primitivo foi introduzido por Chusho [CHU87] e visa determinar um conjunto de arcos do GFC, cuja execução implica na execução de todos os arcos do GFC. No contexto da POKE-TOOL, os arcos são utilizados para descrever os caminhos e associações exigidos pelos critérios Potenciais Usos (PU). Conforme dito anteriormente, com a utilização de arcos primitivos pretende-se otimizar o processo de avaliação da cobertura dos casos de teste e introduzir um mecanismo uniforme para a avaliação da adequação de conjuntos de casos de teste em relação a critérios de teste estruturais de uma maneira geral. O algoritmo utilizado para calcular esses arcos — Algoritmo de Redução de Herdeiros de Fluxo de Dados — é baseado no algoritmo proposto por Chusho [CHU87], mas ligeiramente modificado para se adequar aos requisitos dos critérios baseados em fluxo de dados, em particular, dos critérios PU.

Outra atividade é determinar o *grafo def*; para determiná-lo é necessário fazer análises sintática e semântica dos comandos que compõem cada nó do GFC para identificar quais variáveis estão sendo definidas nestes nós. O *grafo def* consiste na extensão do grafo de programa realizada de acordo com os conceitos e diretrizes discutidos na Seção 4.3. Durante a determinação do grafo def é produzida a unidade modificada para teste onde estão inseridas as pontas de prova que registram o caminho percorrido pelo caso de teste. Com o grafo def é possível determinar os *grafo(i)* para cada nó do GFC que possui definição de variáveis. A partir dos grafo(i) e dos arcos primitivos, são obtidos os caminhos e as associações requeridos pelos critérios PU, bem como os descritores utilizados na avaliação.

O módulo **pokernel** é um programa executável que recebe os arquivos mencionados acima como entrada. O **pokernel** é também dividido em sub-módulos que realizam as diversas atividades descritas acima. O grafo de chamada apresentado na Figura 5.4 ilustra os sub-módulos do módulo **pokernel** e os arquivos que cada um gera. Esses sub-módulos são descritos sucintamente a seguir.

Sub-módulo **cal_prim**: calcula os arcos primitivos utilizando o algoritmo de *redução de herdeiros de fluxo de dados* proposto na Seção 4.4.1.

Sub-módulo **parserli**: A função desse sub-módulo é gerar o *grafo def* e a *unidade instrumentada*. A realização dessas duas tarefas depende essencialmente de uma análise sintática e semântica do código fonte. Essa análise, no contexto da POKE-TOOL, é simplificada porque a unidade em teste possui uma representação especial que é o programa em LI.

Determinação do grafo def: Para gerar o grafo def é preciso fazer análises sintática e semântica específicas para o código fonte da unidade em teste; a tarefa básica é identificar as variáveis que são definidas nos comandos associados a um dado nó. Essa tarefa, aparentemente simples, envolve a identificação das construções que provocam uma definição de variável na linguagem da unidade em teste e das variáveis, propriamente ditas, envolvidas nessas construções. Portanto, essa tarefa requer um conhecimento da semântica da linguagem de programação das unidades em teste.

A POKE-TOOL possui um analisador léxico e um analisador sintático genéricos que devem ser configurados para realizar essa tarefa. O analisador léxico é baseado na utilização de autômatos finitos [SET81] e o analisador sintático baseia-se na análise sintática descendente genérica através de grafos sintáticos [WIR76]. Esses dois analisadores identificam as variáveis sendo definidas e as associam ao nó correspondente. Para realizar essa tarefa, os analisadores possuem rotinas que manipulam as estruturas de dados que representam o *grafo def*. Como configurar os analisadores léxico e sintático é descrito em [CHA91b].

A configuração para uma linguagem L envolve determinar o modelo de dados (conjunto de construções que causam definição de variáveis) de L segundo

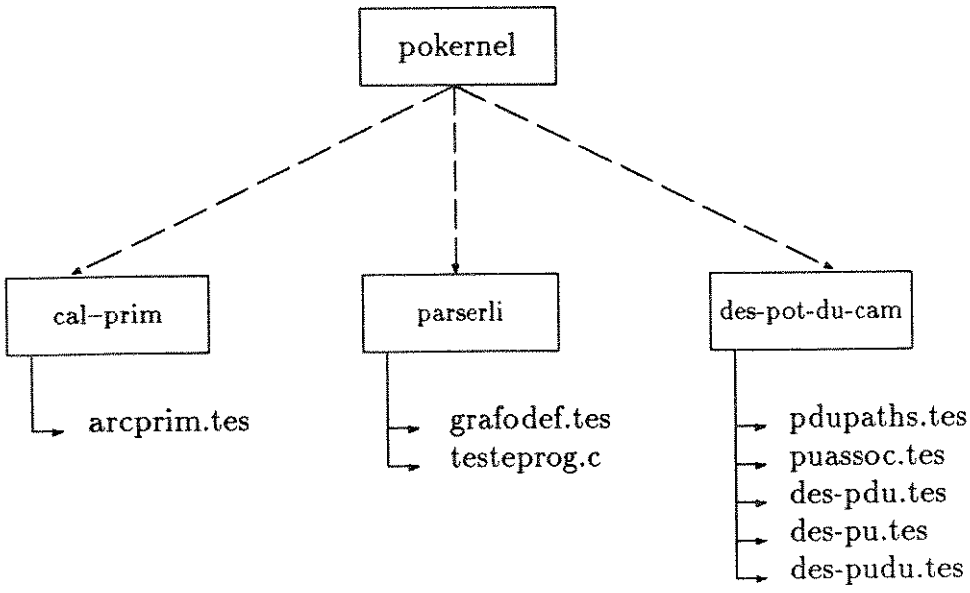


Figura 5.4: Grafo de Chamada do Módulo pokernel.

as diretrizes estabelecidas na Seção 4.3; o tratamento da estrutura de blocos dessa linguagem; e, a configuração dos analisadores léxico e sintático.

A seguir, é apresentado o modelo de dados instanciado para a linguagem C [CHA91a, CHA91b]. Para aplicação dos Critérios Potenciais Usos em programas escritos em C, ou seja, configurar a ferramenta POKE-TOOL para a linguagem de programação C, devem ser considerados praticamente todos os pontos discutidos acima quanto ao modelo de dados, pois esta linguagem possui todas as características abordadas, a menos da passagem de parametros por nome. A linguagem C permite o aninhamento de blocos, a definição de variáveis no início de blocos, suporta o uso de todos os tipos de variáveis simples e compostas e de ponteiros. A Figura 5.5 resume o modelo de dados assumido para a linguagem C; discussões mais detalhadas sobre este modelo podem ser obtidas em [CHA91a, CHA91b].

Geração da Unidade Instrumentada:

O processo de inserir pontas de prova na unidade em teste, gerando a unidade modificada para teste denominada *unidade instrumentada, é chamado Instrumentação*. Esse processo gera, então, uma versão da unidade em teste preparada para as atividades de teste e depende essencialmente da identificação dos comandos que alteram o fluxo de controle do programa em LI. Este processo constitui-se na inserção de pontas de prova e de alguns comandos na unidade em teste e não na análise dos átomos. Portanto, a Instrumentação vai exigir somente a análise sintática da unidade escrita em LI. As pontas de provas (instruções de escrita) geram um "trace" do caminho executado por um dado caso de teste, viabilizando análises posteriores, tal como a análise de cobertura de um dado conjunto de casos de teste. A unidade instrumentada é armazenada no arquivo TESTEPROG.C.

As localizações das pontas de prova dependem do comando da LI a ser instrumentado; em última análise, a localização de uma ponta de prova depende do comando de controle de fluxo da linguagem da unidade que foi traduzida para LI. O modelo de instrumentação adotado é descrito no Capítulo 4.

No Apêndice C, é apresentado o arquivo TESTEPROG.C que contém a unidade instrumentada para o exemplo ENTAB.C, do benchmark.

Sub-módulo **des_pot_du_cam**: O módulo **des_pot_du_cam** determina os grafo(i) e, através desses, os caminhos e associações requeridos e os descritores utilizados para avaliar a adequação de conjuntos de casos de teste em relação aos critérios PU.

A seguir cada uma dessas tarefas é descrita sucintamente.

Geração dos Grafo(i):

Para calcular os grafo(i) de um programa utiliza-se um algoritmo iterativo [MAL88b]. Esse algoritmo utiliza estruturas de dados simples (basicamente *listas ligadas* e *pilhas*) e é de fácil programação.

	DEFINIÇÃO	DEFINIÇÃO POR REFERÊNCIA
VARIÁVEIS COMUNS (SIMPLES)	A asgnop exp. onde asgnop é um comando de atribuição e exp é uma expressão. ⇒ definição de A	fc(&A, ..., ...) onde fc é o nome de uma função ⇒ definição por referência de A.
VARIÁVEIS ESTRUTURADAS	$A[exp_1]$ asgnop exp_2; onde exp_1 e exp_2 são expressões ⇒ definição de A.	fc (A, ..., ...) ou fc (&A[exp_1], ...) ⇒ definição por referência de A.
REGISTROS	A. campo asgnop exp; onde campo é um campo de A. ⇒ definição de A.	fc(& A. campo, ...) ou fc(&A, ...) ⇒ definição por referência de A.
PONTEIROS	A asgnop exp; ⇒ definição de A *A asgnop exp; ⇒ essa definição não é tratada.	fc(&A,...) ⇒ definição por referência de A.

Figura 5.5: Modelo de Dados da Linguagem C para Obtenção do Grafodef

A determinação dos *potenciais-du-caminhos* a partir de um *grafo def* é um problema de análise de fluxo de dados que pode ser reduzido a uma *estrutura monotônica de análise de fluxo de dados* [CHA89]. O fato do cálculo dos potenciais du-caminhos de um grafo def possuir essa propriedade implica que é possível utilizar uma série de algoritmos da literatura [KAM77, HEC77, HOR87] para resolvê-lo.

A utilização desses algoritmos para determinar os potenciais du-caminhos de um grafo def possui algumas vantagens. A principal delas é que se tratam de algoritmos rigorosamente especificados e, portanto, tem-se maior confiança no resultado. Porém, eles possuem duas desvantagens: as estruturas de dados para manipulação de potenciais du-caminhos, nesses algoritmos, podem vir a consumir muito espaço de memória; além de serem complicadas, o que torna difícil a programação de qualquer dos algoritmos. Outra desvantagem desse enfoque é que ele determina somente os requisitos de teste dos critérios todos-potenciais-du-caminhos; os demais critérios não são abordados, acarretando esforço computacional adicional para os outros dois critérios [CHA91b].

Na implementação da POKE-TOOL, utilizou-se o conceito de grafo(i) e arcos primitivos (incluído os arcos p_ϕ) como modelo para determinação dos requisitos de teste dos critérios PU. Nos grafo(i) temos não só os potenciais du-caminhos do critério todos-potenciais-du-caminhos, mas também os arcos primitivos utilizados nas associações dos critérios todos-potenciais-usos e todos-potenciais-usos/du. Dessa maneira, o esforço computacional envolvido no cálculo dos grafo(i) é utilizado pelos três critérios PU.

Em termos de custo computacional (somente tempo de execução sem considerar o consumo de memória), os dois enfoques são equivalentes; para os algoritmos contidos em [KAM77, HEC77, HOR87] e para o algoritmo de cálculo dos grafo(i) este custo é proporcional ao número máximo de potenciais du-caminhos que, no pior caso, é $(11/2)t2^t + 92^t - 10t - 9$ [MAL88b], onde t é o número de decisões do programa [CHA89].

Resumindo, foi utilizado o algoritmo de cálculo dos grafo(i) porque (1) os conceitos de grafo(i) e arcos primitivos são mais adequados aos critérios PU; (2) este algoritmo possui estruturas de dados mais simples e econômicas; (3) é fácil de programar; e (4) possui o mesmo custo computacional que dos demais.

Geração dos Descritores, Caminhos e Associações:

Terminado o processo de geração dos grafo(i), têm-se todas as informações necessárias para a geração dos descritores, caminhos e associações requeridas pelos critérios PU. Esses descritores são especificados na Seção 4.4.1.

Os descritores gerados para o critérios todos-potenciais-du-caminhos, todos-potenciais-usos e todos-potenciais-usos/du são armazenados nos arquivos DES_PDU.TES, DES_PU.TES e DES_PUDU.TES, respectivamente. Os caminhos requeridos pelos critérios todos-potenciais-usos são armazenados no arquivo PDUPATHS.TES e as associações requeridas pelos critérios todos-

potenciais-usos e todos-potenciais-usos/du são armazenados no arquivo PU-ASSOC.TES. No Apêndice C, temos os arquivos descritos acima gerados para o exemplo ENTAB.C.

5.2.7 Módulo Avaliador

O modulo Avaliador faz a análise de adequação de um dado conjunto de casos de teste em relação a um critério PU préviamente selecionado. O **avaliador** da POKE-TOOL recebe como entradas um arquivo com os caminhos ou associações requeridos pelo critério selecionado (PDUPATHS.TES, ou PUASSOC.TES), um arquivo com os descritores desse critério (DES_PDU.TES, DES_PU.TES ou DES_PUDU.TES) e o arquivo que contém o caminho executado pelo caso de teste (PATH.TES). Como saída, o **avaliador** gera um arquivo com os caminhos ou associações que restam ser executados (PDUOUTPUT.TES, PUOUTPUT.TES ou PUDUOUTPUT.TES), um arquivo com os caminhos ou associações efetivamente executados (EXEC_PDU.TES, EXEC_PU.TES ou EXEC_PUDU.TES) e um arquivo que contém os descritores exercitados pelo caso de teste (PDUHIS.TES, PUHIS.TES ou PUDUHIS.TES). Estes últimos arquivos servem como um histórico da avaliação de um dado critério, pois a avaliação de um novo caso de teste vai utilizar estes arquivos para saber quais os descritores que já foram satisfeitos em avaliações anteriores. Os arquivos que contêm o resultado da avaliação (por exemplo, os arquivos PUOUTPUT.TES e EXEC_PU.TES para o critério todos-potenciais-usos) apresentam dois percentuais de *cobertura* para o critério selecionado, conforme descrito na Seção 4.4.2. Conforme definido na Seção 4.4.2, os descritores dos caminhos e associações requeridos pelos critérios PU são autômatos finitos que devem ser satisfeitos para que o caminho ou a associação seja considerada exercitada. Para exercitar um descritor de caminho ou associação é necessário que exista um *caminho*, dentro de um caminho completo, que realize as transições do *estado inicial* até o *estado final* do autômato. O algoritmo de avaliação faz a análise de todos os autômatos simultaneamente através de uma única passada pelo arquivo com o "trace" do caso de teste. É importante observar que o *algoritmo de avaliação* trata descritores, não associações ou caminhos. Depois de dar uma passada pelo arquivo PATH.TES, têm-se alguns autômatos no estado final e outros não. Aqueles que estão no estado final indicam que os caminhos ou associações associados a eles foram executados pelo caso de teste.

5.3 Aspectos de Configuração da POKE-TOOL

Um dos requisitos de projeto da POKE-TOOL é que ela deveria ser uma ferramenta multilinguagem. A POKE-TOOL pode ser configurada fácilmente para testar programas escritos em linguagens procedurais com gramáticas livre de contexto que satisfaçam algumas limitações [CHA91b]. A seguir, é apresentada uma síntese deste processo de instanciação da ferramenta para uma linguagem específica contido em [CHA91b].

Na proposição e projeto da POKE-TOOL estabeleceu-se uma distinção entre as funções dependentes do código fonte das demais, permitindo que essas tarefas sejam isoladas em módulos distintos dos módulos que realizam tarefas independentes da linguagem. Isto permite que os módulos responsáveis por essas atividades possam ser reutilizados nas configurações da ferramenta para as diversas linguagens. Notadamente, o *Cálculo dos Arcos Primitivos*, a *Geração dos Grafo(i) e Descritores* e o mecanismo de *Avaliação* utilizado são completamente independentes da linguagem fonte.

De qualquer maneira, as informações dependentes da linguagem fonte são necessárias. Para obtê-las é necessário produzir um programa semelhante a um "front end" de compilador; este "front end" pode ser genérico ou específico para a linguagem fonte a ser tratada. O "front end" genérico é obtido através de algoritmos que são configurados para tratar os aspectos léxicos, sintáticos e semânticos específicos da linguagem em questão.

A POKE-TOOL utiliza o enfoque genérico. A razão para essa opção foi privilegiar a reutilização de código (e esforço) em detrimento da otimização. Não obstante a existência de soluções otimizadas e, portanto, melhores sob o ponto de vista de eficiência, para o enfoque específico, os algoritmos genéricos de análise sintática e léxica utilizados pela POKE-TOOL têm uma complexidade ($n \log n$, onde n é o comprimento da cadeia de entrada) que não constitui um gargalo no desempenho da ferramenta; este fato foi verificado, na prática, através da utilização da ferramenta na na aplicação do benchmark, descrito no capítulo seguinte.

As atividades da POKE-TOOL relacionadas com a análise estática do código fonte são intrinsecamente dependentes da linguagem fonte da unidade. Retomando a Figura 5.1, observamos que, das funções discriminadas, as seguintes estão relacionadas com a análise do código fonte: *Grafo de Fluxo de Controle*, *Extensão do Grafo de Fluxo de Controle* e *Instrumentação*.

A função *Grafo de Fluxo de Controle* determina o GFC da unidade em teste; é sub-dividida em duas partes: a primeira realiza o mapeamento da unidade em teste para a unidade em teste em LI; a segunda parte realiza a geração do GFC a partir da unidade em LI. Aparentemente, esta função não seria extremamente dependente da linguagem de programação da unidade em teste; entretanto, essa dependência se restringe ao mapeamento para a LI, pois a geração do GFC é genérica.

A *Extensão do Grafo de Fluxo de Controle* consiste em gerar o grafo def. Para gerar o grafo def é necessário que esteja definido o *Modelo de Dados* para a linguagem da unidade em teste. Para a definição do Modelo de Dados existem diretrizes gerais (ver Seção 4.3), mas o modelo em si é específico para uma dada linguagem. Adicionalmente, essa função determina que variáveis são *definidas* em um dado nó, o que implica na análise de cada comando que compõe um nó do GFC.

A *Instrumentação* também é uma função cuja dependência da linguagem de programação se restringe basicamente ao mapeamento para a LI, pois as localizações das pontas de prova colocadas na versão modificada dependem dos comandos da LI (ver

Seção 4.2); ou seja, uma vez realizado o mapeamento já se sabe onde inserir as pontas de prova. Obviamente, é necessária a inserção de comandos para a definição das pontas de prova, declarações de variáveis auxiliares e comandos para a manipulação de arquivos; todos esses comandos são absolutamente vinculados à linguagem de programação da unidade em teste. Porém, onde inserir esses comandos é determinado pela LI, e as modificações a serem feitas consistem somente em alterar os comandos a serem inseridos de uma linguagem para outra.

As demais funções contidas na Figura 5.1 não dependem da linguagem de programação da unidade: o *Cálculo dos Arcos Primitivos* utiliza somente o GFC; a *Geração dos Grafo(i) e Descritores* depende apenas do grafo def; a *Avaliação* utiliza o caminho percorrido pelos casos de teste e os descritores.

Como se vê da discussão acima, identificam-se duas classes de tarefas da POKE-TOOL bem distintas: as *dependentes* e as *independentes* da linguagem da unidade em teste. Este fato vem a confirmar possibilidade de reutilização de partes da POKE-TOOL nas suas diversas configurações.

Basicamente, para configurar a ferramenta, é necessário que o configurador especifique o analisador léxico e o analisador sintático para a linguagem alvo. Em [CHA91c] é descrito como realizar estas tarefas e a instanciação para a Linguagem C; uma síntese desse processo é fornecida a seguir:

1. Configurar o módulo li para realizar a tradução da linguagem de programação alvo para a LI [CAR91].

2. Configurar o analisador léxico da POKE-TOOL. Essa tarefa demanda que o usuário conheça bem os aspectos léxicos da linguagem alvo e está dividida em algumas sub-tarefas.

   2.1. Desenvolver um autômato finito [SET81] que realize a análise léxica da linguagem.

   2.2. Identificar as ações semânticas associadas às transições do autômato finito.

   2.3. Transcrever esse autômato para a notação aceita pelo analisador léxico genérico da POKE-TOOL.

   2.4. Complementar o analisador léxico através de rotinas ("ações semânticas") que realizem a separação dos átomos da linguagem fonte, colocando-os nas estruturas de dados da POKE-TOOL utilizadas para armazená-los.

   2.5. Depurar o analisador léxico conjuntamente com as ações semânticas desenvolvidas.

3. Configurar o analisador sintático da POKE-TOOL. Esta tarefa demanda que o usuário conheça bem a sintaxe e a semântica da linguagem alvo e está dividida em algumas sub-tarefas.

3.1. Descrever a gramática da linguagem fonte na forma de *grafo sintáticos* [WIR76].

3.2. Identificar os pontos onde devem ser ativadas as rotinas semânticas nos grafos sintáticos.

3.3. Transcrever os grafos sintáticos para a notação aceita pela POKE-TOOL.

3.4. Complementar o analisador sintático com rotinas semânticas que realizem a identificação das variáveis definidas, ajustando os campos do vetor **info_no**.

3.5. Depurar o analisador sintático conjuntamente com as rotinas desenvolvidas.

4. Ajustar os procedimentos que inserem código fonte na nova versão da unidade em teste com as instruções da nova linguagem de programação aceita pela POKE-TOOL.

5. Compilar e ligar os arquivos que constituem os analisadores léxico e sintático genéricos, as ações e rotinas semânticas, e os procedimentos de inserção de código fonte atualizados com os demais arquivos que constituem a POKE-TOOL.

A POKE-TOOL possui uma biblioteca de rotinas para manipulação das estruturas de dados utilizadas. Essas rotinas e as estruturas de dados estão detalhadas em [CHA91a].

5.4 Considerações Finais

Apresentou-se a arquitetura de uma ferramenta de teste de apoio aos critérios Potenciais Usos. Na realidade, esta ferramenta visa a dar suporte à atividade de teste estrutural de programas, ou seja, não se pretendeu vinculá-la a um critério espeçífico. Outros critérios podem ser facilmente incorporados nesta ferramenta. Um objetivo de mais longo prazo é obter um ambiente de suporte integrado às atividades de teste, depuração e manutenção, uma vez que as informações pertinentes a essas atividades são fortemente relacionadas.

A ferramenta POKE-TOOL, proposta nesse trabalho, possui as seguintes características: é uma ferramenta de suporte à aplicação de um conjunto de critérios fazendo basicamente análise de cobertura para um conjunto de casos de teste. Ela fornece uma interface simples para o usuário (sem maiores grafismos) mas fornece uma organização dos dados de teste que só é encontrada nas ferramentas comerciais. Leva em conta considerações inter-procedurais, que tornam o teste com a POKE-TOOL mais exaustivo e abrangente do que o teste com a maioria das ferramentas; e permite que o usuário a configure para novas linguagens, que é uma característica muito interessante dada a enorme gama de linguagens em uso hoje em dia.

As ferramentas comerciais possuem a sofisticação adicional de possibilitar a geração do programa executável automaticamente. A única ferramenta baseada em fluxo de

dados que realiza tal tarefa é ASSET, que necessita de um arquivo *makefile* para realizá-la. Uma facilidade interessante seria a geração de programas "drivers" e "stubs" para auxiliar o teste de unidades; entretanto, nenhuma das ferramentas estudadas possuem essa capacidade. Pretende-se inserir essas características na POKE-TOOL. Outro aspecto, que está em desenvolvimento, é a facilidade para tratamento de caminhos não executáveis [VER91a, VER91B].

Deve-se ressaltar que, com a implementação e utilização da POKE-TOOL, os modelos propostos no Capítulo 4, foram de certa forma consolidados e validados.

Capítulo 6

Critérios Potenciais Usos: Aplicação de um Benchmark

A falta de planejamento dos recursos necessários para o desenvolvimento de software tem sido apontado como um dos motivos da crise de software. O planejamento das atividades de teste deve fazer parte do planejamento global do sistema; a falta de tempo e de recursos humanos capacitados, e a indisponibilidade de ferramentas adequadas são os principais problemas enfrentados pelas equipes de teste, decorrentes da falta de planejamento e consequentemente de um plano de teste.

Modelos para estimar o esforço das atividades de teste são iniciativas que contribuem e motivam uma prática mais sistemática e disciplinada no desenvolvimento de um produto de software. A quantificação do esforço de teste pode então ser incorporada em planos de teste e auxiliar nas atividades de garantia de qualidade. Esta prática, em geral, leva a um aprimoramento da qualidade global do produto e o custo de atividades de outras fases do desenvolvimento pode ser sensivelmente reduzido; por exemplo, o custo do teste de regressão na fase de manutenção.

Segundo Ntafos [NTA88], um fator comum no custo das atividades de teste — geração e execução dos casos de teste, e análise dos resultados — é o número de casos de teste necessários para satisfazer cada critério. A complexidade do critério fornece um limitante superior mas, do ponto de vista prático, é de fundamental importância a realização de experimentos que forneçam indicações da média do número de casos de testes para classes de programas mais usuais.

Conforme apontado no Capítulo 2, estudos empíricos têm sido conduzidos com o objetivo principal de investigar o custo do uso de critérios de testes baseados em fluxo de dados nas atividades de teste (por exemplo, vide [WEY90, WEY88b, BIE89]). O conjunto de programas utilizado por Weyuker [WEY90] — extraído do livro de Kernighan & Plauger[KER81] — tem sido encarado como um benchmark [BEI90]. Observou-se ainda que um outro objetivo desses estudos é determinar uma maneira de estimar o número de casos de testes necessários para satisfazer um dado critério

para um dado programa P a ser testado. Weyuker [WEY90] observou que para a Família de Critérios de Fluxos de Dados [RAP85], o estudo empírico indicou que o número de casos de teste requeridos para satisfazer os critérios dessa família pode ser visto como linear em função do número de comandos de decisão do programa. Esta conclusão é suportada principalmente pelo resultado da regressão linear entre o número de casos de teste — *variável resposta* — e o número de comandos de decisão — *variável independente* . O cálculo da média ponderada da razão entre o número de casos de teste suficientes para satisfazer o critério selecionado e o número de comandos de decisão também contribuiu nessa direção. Outro ponto importante é a caracterização do pior caso empírico, denominado *complexidade empírica*: o máximo valor obtido dentre os quocientes do número de casos de teste pelo número de comandos de decisão. Isto é uma boa indicação de que os critérios baseados em fluxo de dados são utilizáveis do ponto de vista prático, pois podem ser vistos como lineares no número de comandos de decisão, até mesmo no pior caso empírico [WEY90].

Neste capítulo são descritas a realização e análise da aplicação dos Critérios Potenciais Usos, utilizando-se a ferramenta de teste POKE-TOOL, no mesmo benchmark extraído de [KER81]. São três os objetivos principais:

i) Contribuir para demonstrar a factibilidade dos critérios baseados em análise de fluxo de dados, em particular dos Critérios Potenciais Usos;

ii) Determinar modelos para estimar, nas diversas etapas de desenvolvimento do software, o número de casos de teste necessários para testar um dado programa com os critérios Potenciais Usos.

iii) Explorar a influência de outras características do produto em teste, além do número de comandos de decisão, na determinação dos modelos de estimativas. Por exemplo, seria de se esperar que o número de variáveis utilizadas no programa fosse significativo na estimativa do número de casos de teste necessários para satisfazer os critérios baseados em análise de fluxo de dados, em particular para os critérios Potenciais Usos.

Na Seção 6.1 são descritas a realização e a coleta dos resultados, base para a análise apresentada na Seção 6.2; nesta são discutidos e analisados diversos modelos para a estimativa do número de casos de testes requeridos pelos critérios Potenciais Usos e comparados os resultados com os obtidos para a Família de Critérios de Fluxo de Dados. Ainda, na Seção 6.2 são fornecidos resultados indicativos da influência das variáveis de controle consideradas, na estimativa do número de caminhos não executáveis e são levantadas algumas conjecturas sobre o estabelecimento de métricas de complexidade de software.

6.1 Realização e Coleta de Resultados

Neste experimento procurou-se explorar, além da influência do número de comandos de decisão na estimativa do número de casos de teste, a relação entre a ocorrência de variáveis no programa e o número de casos de teste requeridos para satisfazer os critérios baseados em análise de fluxo de dados, em particular os critérios Potenciais Usos. Um resultado secundário, do ponto de vista do objetivo principal do experimento, é determinar a influência dessas variáveis de controle no número de caminhos e associações não executáveis.

Deve-se salientar que são vários os pontos que influenciam o número de casos de teste requeridos por um critério baseado em fluxo de dados; por exemplo, o número de comandos de decisão tem uma forte influência, assim como as definições e (potenciais) usos ao longo do programa. Outros pontos são de relevância, tais como o tipo de comando (if, while, ...), a ordem de ocorrência dos comandos, o número de definições e a seqüência de suas ocorrências, entre outros. Por exemplo, pode-se elaborar programas com distribuição adequada de ocorrências de variáveis com o objetivo de maximizar o número de elementos requeridos, assim como, pode-se distribuir estas ocorrências de forma a minimizar esse número. Esses fatores foram considerados na análise de complexidade dos Critérios Potenciais Usos [MAL88b, MAL91a] e têm forte influência nas estimativas realizadas; sem dúvida, são explicações para a variabilidade dos modelos apresentados, pois as variáveis de controle utilizadas na determinação dos modelos não captam todos esses aspectos.

Intuitivamente, espera-se que o uso de variáveis tenha uma forte influência no número de casos de teste requeridos, uma vez que os caminhos e associacões requeridos são extremamente dependentes do fluxo de dados; mais ainda, na determinação da complexidade dos critérios, a ocorrência e a distribuição das definições das variáveis do programa foi um fator relevante. Neste sentido, serão observados o número de variáveis utilizadas no programa, o número de definições de variáveis, assim como o número de nós com definição de variáveis. A seguir, são descritos a realização do experimento e a organização e coleta dos resultados obtidos.

6.1.1 Estratégia adotada para a condução do Benchmark

Inicialmente, para cada um dos programas que compõem o benchmark, determinaram-se os seguintes dados (*variáveis de controle*), sintetizados na Tab. 6.1:

- número de comandos de decisão (C1).
- número de variáveis utilizadas (C2).
- número de definições de variáveis (C3).

Tabela 6.1: Variáveis de Controle Relativas ao Benchmark

UNIDADE	# Comandos de Decisão / # nós (C1/C5)	# Variáveis Utilizadas (C2)	# Definições de variáveis (C3)	# Nós com Definição de Variáveis (C4)
DODASH	6/19	7	8	3
ARCHIVE	6/18	4	9	6
RQUICK	6/14	11	14	6
GETFNS	6/17	6	13	9
COMPARE	6/14	9	16	5
ENTAB	6/15	4	10	8
EXPAND	6/18	2	6	5
CMP	5/16	3	5	2
COMPRESS	5/16	3	6	5
UNROTATE	7/20	5	21	13
TRANSLIT	12/30	8	22	12
COMMAND	15/19	29	44	15
GETCMD	14/43	2	16	15
AMATCH	7/22	6	19	12
OMATCH	15/29	5	12	8
GTEXT	3/9	9	13	5
GETDEF	9/26	7	12	7
GETONE	9/22	7	8	3
GETNUM	6/19	4	4	1
MAKEPAT	10/30	8	22	13
SPREAD	5/14	9	18	8
CHANGE	5/11	4	12	7
GETFN	5/13	5	7	4
SUBST	9/27	17	36	16
EDIT	9/22	7	16	10
GETLIST	6/16	9	19	8
APPEND	5/16	6	12	8
CKGLOB	6/19	6	11	8
OPTPAT	5/15	3	5	3

- número de nós com definição de variáveis (C4).
- número de nós do grafo de programa (C5).

Para a realização do benchmark propriamente dito, ou seja, elaboração dos conjuntos de casos de testes (CCT) e análise de adequação dos conjuntos de casos de teste em relação a alguns dos critérios Potenciais Usos — *todos-potenciais-usos, todos-potenciais-usos/du e todos-potenciais-du-caminhos* — adotaram-se, a exemplo de [WEY90], algumas diretrizes, concretizadas nas etapas descritas a seguir, que deverão constituir uma conduta padrão em experimentos semelhantes que venham a ser realizados posteriormente.

1) Leitura da descrição funcional da unidade, de detalhes e alternativas que nortearam a implementação das unidades a serem testadas.

2) Elaboração do conjunto inicial de casos de teste (CICT). A minimização dos conjuntos de casos de teste não é almejada. São selecionados casos de teste típicos da aplicação, atômicos, i.e., com propósito único, ao invés de selecionarem-se casos de teste que cobrissem várias características simultaneamente. Este conjunto foi derivado essencialmente a partir de exemplos, sugestões e considerações de Kernighan & Plauger [KER81], de maneira informal e não sistemática; procurou-se identificar os casos especiais e valores limites das condições de entrada e saída; no entanto, não ficou caracterizada a aplicação explícita de nenhum critério de teste funcional, tais como, "Equivalence Partitioning" ou "Boundary Value Analysis". Este experimento está sendo repetido aplicando-se de forma sistemática os dois critérios citados acima para a elaboração do CICT.

3) Implementação da unidade a ser testada utilizando a linguagem de programação *C*, uma vez que a versão atual da POKE-TOOL suporta apenas esta linguagem.

4) Submissão do conjunto CICT à unidade a ser testada, sob o controle da POKE-TOOL. Quando da existência de dúvida quanto à saída esperada, a versão PASCAL da unidade foi executada e as dúvidas dirimidas; esta situação raramente ocorreu. Neste sentido aplicou-se uma variante de "Back to Back Testing" [VOU90].

5) Análise de adequação do CICT em relação aos critérios Potenciais Usos (PU).

6) Geração e submissão de novos casos de testes a partir de informações oriundas da análise de cobertura feita pela POKE-TOOL, com o objetivo de cobrir as associações requeridas pelo critério todos-potenciais-usos. Casos de testes são gerados até que todas as associações executáveis requeridas sejam exercitadas. Se o conjunto de caminhos e associações ainda a serem exercitados for composto somente de caminhos não executáveis, dá-se por encerrado o teste da unidade de acordo com o critério em questão; neste caso, diz-se que o Conjunto de Casos de Teste quase satisfaz ("almost satisfies") o critério [WEY90]. A determinação

da executabilidade de um caminho ou associação é indecidível, e é de responsabilidade única do testador, uma vez que a POKE-TOOL, na versão atual, não fornece suporte nesta direção. O Conjunto de Casos de Testes nesta etapa é denominado CFCT/PU.

7) Geração e submissão de novos casos de testes a partir de informações oriundas da análise de cobertura feita pela POKE-TOOL, com o objetivo de cobrir associações requeridas pelo critério todos-potenciais-usos/du ainda não exercitadas. Na existência de associações não executáveis o procedimento adotado é o mesmo descrito no item 5). O Conjunto de Casos de Testes nesta etapa é denominado CFCT/PUDU.

8) Geração e Submissão de novos Casos de Testes a partir de informações oriundas da análise de cobertura feita pela POKE-TOOL, com o objetivo de cobrir os caminhos requeridos pelo critério todos-potenciais-du-caminhos ainda não exercitados. Na existência de associações não executáveis o procedimento adotado é o mesmo descrito no item 5). O Conjunto de Casos de Testes nesta etapa é denominado CFCT/PDU.

9) Coleta e Organização de Informações para a Análise dos resultados do benchmark.

Como resultado imediato da realização do benchmark, os seguintes resultados, sintetizados na Tabela 6.2, foram obtidos:

- Conjunto Inicial dos Casos de Testes (CICT) e a análise da adequação do CICT para cada um dos critérios, ou seja, a análise de cobertura em relação a cada um dos critérios Potenciais Usos.
- Conjunto de Casos de Testes para o Critério CR1 (CFCT/PU)
- Conjunto de Casos de Testes para o Critério CR2 (CFCT/PUDU)
- Conjunto de Casos de Testes para o Critério CR3 (CFCT/PDU)
- Conjunto de associações e caminhos não executáveis para os critérios Potenciais Usos.

Tabela 6.2: Variáveis Respostas e Associações Requeridas Relativas ao Benchmark

UNIDADE	Card. CICT	Cardinalidade CFCT			# Assoc. não exec./requeridas		
		CR1	CR2	CR3	CR1	CR2	CR3
DODASH	7	15	15	15	0/33	0/33	0/21
ARCHIVE	7	7	7	7	0/17	0/17	0/17
RQUICK	1	2	2	2	6/55	19/55	33/51
GETFNS	5	7	7	7	8/44	14/44	9/34
COMPARE	10	11	12	12	0/38	6/38	30/64
ENTAB	8	14	14	14	23/80	31/80	41/70
EXPAND	12	12	13	13	14/39	15/39	10/25
CMP	7	7	7	7	3/13	3/13	3/12
COMPRESS	12	14	14	14	3/39	6/39	10/34
UNROTATE	3	6	6	6	14/94	39/94	29/70
TRANSLIT	33	37	38	48	6/138	6/138	203/333
COMMAND	15	15	15	15	3/47	3/47	19/61
GETCMD	15	15	15	15	0/119	0/119	0/119
AMATCH	9	14	14	14	70/112	70/112	61/87
OMATCH	13	13	13	13	12/33	12/33	25/43
GTEXT	5	5	5	5	6/33	11/33	10/22
GETDEF	13	20	23	23	3/59	9/59	9/49
GETONE	8	12	14	20	3/62	6/62	147/177
GETNUM	7	8	8	8	0/8	0/8	5/12
MAKEPAT	36	39	39	39	73/207	87/207	169/253
SPREAD	7	12	17	17	7/61	10/61	14/47
CHANGE	8	8	8	8	1/38	1/38	2/27
GETFN	7	7	7	7	2/18	2/18	7/20
SUBST	10	29	26	28	43/219	74/219	235/322
EDIT	17	32	40	40	20/130	21/130	261/357
GETLIST	6	13	14	15	20/82	22/82	64/102
APPEND	7	7	7	7	11/49	12/49	22/43
CKGLOB	7	7	7	7	1/35	11/35	7/27
OPTPAT	5	5	5	5	1/13	1/13	13/21

6.2 Análise dos Resultados do Benchmark

Alguns pontos devem ser ressaltados quanto aos programas testados e quanto aos resultados obtidos antes de proceder-se à análise propriamente dita desses resultados. Uma consideração importante é que dois dos programas testados — RQUICK e AMATCH — são recursivos e a atual versão da POKE-TOOL não possibilita um tratamento uniforme de procedimentos recursivos e procedimentos não recursivos [MAL91c]. A análise foi então conduzida para dois conjuntos de pontos distintos — um conjunto incluindo procedimentos recursivos, totalizando 29 pontos observados, e o outro excluindo os procedimentos recursivos, com 27 pontos.

Alguns procedimentos apresentaram maior dificuldade de serem testados do que outros; por exemplo, os programas COMMAND, GETCMD e OMATCH demandaram cerca de t casos de teste, onde t é o número de comandos de decisão. Estes programas apresentam complexidade ciclomática 16, 15, 16, respectivamente e contêm estruturas de controle CASE ou ELSEIF o que constitui, segundo McCabe, uma explicação para este fato [McC76].

Outros programas, de complexidade ciclomática da mesma ordem, apresentaram dificuldades para a elaboração de conjuntos de casos de teste que satisfizessem os critérios Potenciais Usos; por exemplo, os programas MAKEPAT, SUBST e TRANSLIT demandaram cerca de $4 * t$ casos de teste. Estes resultados são coerentes com os resultados discutidos por McCabe [McC76].

Dois outros programas — EDIT e SPREAD, de complexidade ciclomática menor que dez, apresentaram dificuldades de serem testados, contrariando a observação de McCabe de que programas com complexidade ciclomática menor que 10 (dez) são mais fáceis de serem testados do que programas com complexidade ciclomática maior do que dez.

Uma possível explicação é que os programas que apresentaram maior dificuldade, citados acima, contêm estruturas de controle similares à estrutura de controle que maximiza o número de potenciais-du-caminhos [MAL88b, MAL91a].

Outro programa que merece destaque é o UNROTATE, de complexidade ciclomática oito, pois exigiu menos do que t casos de teste; este programa utiliza 5 variáveis e 21 definições de variáveis e contém 4 laços, o que não favorece a geração de potenciais-du-caminhos.

Para todos os programas citados acima, o número de potenciais-du-caminhos presentes em cada um deles parece constituir uma explicação razoável para as dificuldades identificadas. Todos os demais programas do benchmark demandaram entre t e $3 * t$ casos de teste e, da mesma forma, o número de potenciais-du-caminhos reflete as dificuldades encontradas para o teste destas unidades.

Para cada um dos critérios Potenciais Usos, os seguintes resultados, sintetizados na

Tabela 6.3, foram computados em função dos dados coletados:

1) A média ponderada M da razão entre o número de comandos de decisão t de um dado programa pelo número de casos de teste, nct, suficiente para satisfazer o critério selecionado (*análise do caso médio empírico*); expressou-se então nct em função de M, ou seja, $nct = (1/M)t$, onde M é definido por

$$M = \frac{1}{n} \sum_{i=1}^{n} \frac{t_i}{nct_i}$$

e n é o número de programas considerados.

2) O máximo valor da razão entre o número de comandos de decisão, t, de um dado programa pelo número de casos de teste, nct, suficiente para satisfazer o critério selecionado (*análise do pior caso empírico*).

3) O mínimo valor da razão entre o número de comandos de decisão t de um dado programa pelo número de casos de teste, nct, suficiente para satisfazer o critério selecionado.

A título de comparação, para o critério *todos-du-caminhos* , o critério mais exigente da Família de Critérios de Fluxo de Dados, a complexidade empírica no pior caso, para o benchmark em questão, é *3.67*, ou seja, o número máximo de casos de teste (nct) é igual a *3.67* vezes o número de comandos de decisões (t), expresso na equação $nct = 3.67 * t$; considerando a média ponderada M, temos que $nct = 0.81t$. Para o *critério todos-potenciais-usos, no pior caso empírico,* $nct = 4.45 * t$; considerando a média ponderada M, tem-se $nct = 1.44t$. Deve-se ressaltar que o mesmo programa contribuiu para o pior caso empírico dos três critérios Potenciais Usos; uma das explicações é que este programa contém estruturas de controle similares à da estrutura que maximiza o número de potenciais-du-caminhos.

Esses dados indicam que, do ponto de vista prático, os critérios Potenciais Usos requerem, em geral, um número bastante modesto de casos de teste. Considerando que a complexidade destes critérios é de ordem exponencial e que a satisfação de um critério de teste usualmente é um dos itens dos critérios de encerramento das atividades de teste, esses resultados motivam a realização de outros trabalhos em torno dos critérios Potenciais Usos; em particular, em torno do conceito *potencial uso*, fazendo-se por exemplo, a combinação e o estudo comparativo entre critérios baseados em fluxo de dados.

Objetivando fornecer uma maneira de estimar o esforço necessário para conduzir as atividades de teste, selecionado um critério, vários modelos de estimativas foram explorados, com o apoio do Sistema MINITAB [RYA76] — um Sistema de Computação Estatístico de propósito geral —, utilizando-se as *variáveis de entrada (variáveis independentes ou preditores)* descritas na Tabela 6.1. Duas variáveis respostas ou dependentes foram alvo da análise apresentada: i) o número de casos de teste, representado por nct e ii) o número de caminhos não executáveis, representado por nex.

A Tabela 6.4 apresenta uma síntese dos melhores modelos obtidos para os três critérios Potenciais Usos ásicos. Para uma melhor ilustração, é apresentada, na Tabela 6.5, uma síntese de alguns dos modelos obtidos para estimar o número de casos de teste, para o critério todos-potenciais-du-caminhos, considerando-se procedimentos não recursivos. A primeira coluna dessa tabela é um nome de referência ao modelo propriamente dito, ilustrado na segunda coluna sob o título *Expr.*; a terceira coluna ($R2$) representa o quadrado do coeficiente de correlação amostral r e indica a proporção de variabilidade explicada pelo modelo. A quarta coluna (S), por sua vez, indica o desvio padrão da estimativa e fornece uma indicação do desvio padrão σ. Para o modelo de estimativa do número de casos de teste utilizando-se a variável de controle número de potenciais-du-caminhos é também fornecida a estatística para teste de hipótese relativa a esta variável de controle.

Uma síntese das estatísticas relativas aos parâmetros dos modelos da Tabela 6.5 é apresentada na Tabela 6.6. A primeira coluna dessa tabela indica o nome de referência do modelo; da segunda à quinta coluna são indicadas as estatísticas para teste de hipóteses relativas a cada uma das variáveis de controle dos modelos explorados; as sexta e sétima colunas indicam, respectivamente, o intervalo de validade da variável de controle t do modelo e o número de pontos considerados.

No Apêndice D é apresentada uma visão global da análise estatística realizada, sendo apresentados detalhadamente os resultados obtidos para o critério todos-potenciais-du-caminhos; a análise conduzida para os outros dois critérios Potenciais Usos Básicos foi semelhante ao critério todos-potenciais-du-caminhos.

Na exploração dos modelos para estimativa, os programas (pontos), discutidos no início desta seção, caracterizaram-se como os mais influentes na determinação do modelo mais adequado. Observou-se que alguns destes pontos chegavam até mesmo a mascarar a relevância das variáveis de entrada no estabelecimento do modelo propriamente dito.

Os três pontos com $t \geq 12$ observados, indicam uma tendência de diminuição da relação nct/t à medida que o número de comandos de decisão t aumenta; esperava-se que, em geral, à medida que o tamanho do programa crescesse, o número de casos de teste requeridos crescesse até, eventualmente, numa taxa maior. Esta mesma tendência foi observada nos dados do experimento conduzido por Shimeall e Leveson apresentados por Weyuker [WEY90]. Na realidade, fazendo-se uma regressão linear dos dados de Shimeal e Leveson, observa-se que a significância do parâmetro da variável de controle t é irrelevante, ou seja, o modelo de estimativa é uma constante. No caso específico deste trabalho, a explicação pode-se dar pelo simples fato desses programas serem constituídos de estruturas de controle CASE e ELSEIF (que simula uma estrutura CASE). De um modo geral, uma conjectura na tentativa de explicar este comportamento é que a relação $variv/t$ diminui sensivelmente à medida que t cresce, ou seja, o número de variáveis utilizadas não cresceria na mesma proporção que o número de comandos de decisão; adicionalmente, o número de definições de variáveis poderia crescer e suas distribuições poderiam contribuir no sentido de minimizar a inter-relação entre a estrutura

de fluxo de controle e o fluxo de dados no programa. Isto de certa forma contribuiria para minimizar o esforço das atividades de teste.

Os fatos acima apontam a necessidade de que novos dados sejam obtidos tanto para a classe de programas que caracterizou este benchmark, como para outras classes de programas e áreas de aplicação. Ainda, esses resultados ressaltam a necessidade de uma análise estatística mais minuciosa dos dados obtidos na condução do benchmark, principalmente a realização de uma análise de inferência e o uso de métodos robustos de regressão [ACH90,DEN82] para a escolha de um modelo definitivo de estimativa do número de casos de teste necessários para satisfazer os critérios Potenciais Usos.

Apesar das limitações abordadas acima, dois pontos devem ser ressaltados como consequência dessa análise exploratória. O primeiro refere-se à relevância das variáveis de controle *variav*, *def* e *nodef* na determinação dos modelos de estimativa do número de casos de testes e do número de caminhos não executáveis. Considerando-se os níveis de significância usuais utilizados na realização de análises estatísticas utilizando-se a distribuição t de Student e através de uma análise breve dos resíduos, conclui-se que todas as três introduzem melhoras significativas no modelo que leva em conta somente o número de comandos de decisão e são importantes na determinação do modelo. O número de nós com definição de variáveis (*nodef*), por exemplo, é extremamente relevante na determinação do número de associações não executáveis para o critério CR1, até mesmo mais significativo do que o número de comandos de decisão. O segundo ponto, refere-se à forte correlação do número de potenciais-du-caminhos com o número de casos de testes e com o número de caminhos e associações não executáveis para o critério todos-potenciais-du-caminhos. Isto significa que dado o número de potenciais-du-caminhos a previsão do número de casos de teste e do número de caminhos não executáveis presentes no programa é realizada com uma confiança bastante grande. Por exemplo, para o critério todos-potenciais-du-caminhos, determinaram-se modelos para previsão do número de caminhos não executáveis com r^2 da ordem de 99%. Um fato interessante e que merece um estudo mais detalhado é que o número de potenciais-du-caminhos não apresentou uma correlação muito alta com o número de associações não executáveis para os outros dois critérios.

Obviamente, alguns modelos são mais adequados para as fases iniciais do desenvolvimento do que outros; por exemplo modelos que consideram o número de potenciais-du-caminhos, o número de nós com definição de variáveis ou mesmo o número de definições de variáveis não seriam tão úteis na fase de análise e especificação quanto os modelos que consideram somente o número de comandos de decisão e o número de variáveis como variáveis de controle. É importante observar que, para o benchmark conduzido, os modelos mais adequados para as fases iniciais do desenvolvimento do software fornecem uma maneira de prever a variável resposta com mesmo nível de variabilidade dos modelos mais adequados para as fases posteriores do desenvolvimento; nestas, algumas outras informações já são mais precisamente conhecidas e quantificadas, como o número de potenciais-du-caminhos. Os modelos apropriados para as fases iniciais de desenvolvimento de software são muito importantes para as atividades de planejamento do desenvolvimento.

Deve-se ressaltar que a determinação do número de potenciais-du-caminhos em um dado programa é uma função disponível na POKE-TOOL e reflete, por definição, alguns aspectos do inter-relacionamento da estrutura de controle com as ocorrências de variáveis ao longo do programa. Fazendo-se uma abstração, os potenciais-du-caminhos a partir de um nó i, de uma certa forma, refletem os possíveis efeitos da mudança de estado do programa, decorrentes das definições de variáveis que ocorrem no nó i, nos processamentos subsequentes.

O conceito de *potencial uso* e a consequente definição de *potencial-du-caminho* fornecem, na realidade, uma medida do grau de inter-relacionamento entre a estrutura de controle do programa e o fluxo de dados; isto pode refletir outras propriedades associadas ao programa: dificuldades de testar, número de caminhos não executáveis, dificuldade de depuração e de manutenção, entre outras. Na análise conduzida, observou-se que o número de potenciais-du-caminhos tem uma correlação bastante alta com o número de caminhos não executáveis e com o número de casos de teste necessários para satisfazer os critérios Potenciais Usos, constituindo um indicativo bastante forte da dificuldade de condução das atividades de teste. Esta medida parece ser uma medida robusta que deveria ser utilizada no estabelecimento de métricas e de modelos de previsão. Estes fatos ressaltam que os conceitos básicos que norteiam a definição de um critério de teste são extremamente importantes e que, na realidade, um critério de teste pode ser visto como uma medida de complexidade.

No contexto da POKE-TOOL duas métricas são fornecidas em decorrência da análise de cobertura. Uma delas fornece um percentual de associações exercitadas em relação às requeridas. Por exemplo, para o critério todos-potenciais-du-caminhos é fornecida uma porcentagem dos caminhos executados com relação aos caminhos requeridos. A outra métrica fornece uma média dos percentuais de cobertura dos elementos requeridos em relação a cada grafo(i).

Mais especificamente, seja $DEF = \{i \in N \mid defg(i) \neq \phi\}$, *card(U)* uma função que retorna a cardinalidade do conjunto U, *ASSOCEX(i)* o número de associações exigidas por um grafo(i) executadas, *ASSOCTOT(i)* o número total de associações exigidas por um grafo(i), *ASSOCEX* o número de associações executadas e *ASSOCTOT* o número total de associações requeridas. A primeira métrica, chamada *Cobertura Total* (*CT*), é dada por

$$CT = \frac{ASSOCEX}{ASSOCTOT}.$$

A segunda métrica, chamada *Cobertura de Fluxo de Dados* (*CFD*), é dada por

$$CFD = \frac{\sum_{i \in DEF} \frac{ASSOCEX(i)}{ASSOCTOT(i)}}{card(DEF)}.$$

Malevris [MALE90] propõe uma métrica para determinar a executabilidade de caminhos baseada no número de predicados existentes no caminho; a suposição é de que quanto maior o número de predicados maior a probabilidade de que esse caminho seja não executável. Heurísticas baseadas em uma combinação de técnicas de

avaliação simbólica e análise de fluxo de dados foram propostas para identificação de não executabilidade [FRA87]. Estudos para incorporar na POKE-TOOL facilidades para tratamento de caminhos não executáveis e estudos de comparação entre métricas e heurísticas para previsão e determinação de caminhos não executáveis estão sendo conduzidos [VER91a, VER91b]. Esses estudos contribuirão para a definição de estratégias de geração de casos de teste baseadas em fluxos de dados.

6.3 Considerações Finais

Com a aplicação do benchmark obtiveram-se resultados bastante interessantes. Em geral, pode-se dizer que esses critérios, do ponto de vista prático, são factíveis e demandam um número de casos de teste relativamente pequeno. Foram obtidos vários modelos, bastante razoáveis, para estimar as variáveis resposta: número de casos de teste requeridos (*nct*) e número de caminhos não executáveis (*nex*); identificaram-se modelos adequados para diversas fases de desenvolvimento de software. Outra contribuição nesse sentido foi evidenciar que o uso de informações sobre as variáveis do programa são relevantes na determinação de modelos para estimar estas variáveis respostas; essas informações devem, portanto, ser consideradas em estudos posteriores semelhantes ao apresentado neste trabalho. Um outro resultado foi ressaltar que o *número de potenciais-du-caminhos* é uma medida de complexidade robusta e que reflete características importantes do software para as atividades de teste, depuração e manutenção. Com base nessa métrica, duas medidas disponíveis na POKE-TOOL foram discutidas: Cobertura Total e Cobertura de Fluxo de Dados.

Ressaltou-se que os conceitos básicos que norteiam a definição de um critério de teste são extremamente importantes e que, na realidade, um critério de teste pode ser visto como uma medida de complexidade. Neste sentido, os estudos comparativos e propriedades de critérios de teste podem ser aplicados às medidas de complexidade; por exemplo, a *relação de inclusão* estabeleceria uma ordem parcial entre medidas de complexidade.

Os resultados obtidos com a realização do experimento podem ser considerados bastante promissores e motivadores de novas pesquisas e estudos empíricos em torno dos Critérios Potenciais Usos, principalmente em torno da combinação de conceitos na definição de novos critérios e do estudo comparativo do custo da aplicação desses critérios e de suas adequação a classes de erros. Também motivam a condução de manutenção de aperfeiçoamento na ferramenta POKE-TOOL, visando principalmente incorporar novas facilidades de suporte às atividades de teste, tais como, visualização gráfica, identificação e manipulação de caminhos não executáveis, recursos para manutenção de conjuntos de casos de teste, entre outras.

Tabela 6.3: Valores Médios

	CR1	CR2	CR3
media ponderada	nct = 1.38t	nct = 1.41t	nct = 1.44t
maximo nct/t	3.9	4.45	4.45
minimo nct/t	0.34	0.34	0.34

Tabela 6.4: Melhores Modelos Obtidos

Critério todos-potenciais-usos (CR1)

Número de casos de teste (nct)	Associações não executáveis (nex)
nct = -11.7 + 3.96 t	nex = -25.6 + 5.82 t
nct = -13.4 + 3.85 t + 0.411 variav	nex = -15.5 + 2.46 t + 1.52 variav
nct = -12.6 + 3.64 t + 0.247 def	nex = -11.6 + 1.35 t + 0.979 def
nct = -13.2 + 3.31 t + 0.868 nodef	nex = -25.7 + 2.69 t + 3.03 nodef
nct = 6.90 + 0.0759 potducam	nex = 2.67 + 0.0888 potducam

Critério todos-potenciais-usos/du (CR2)

Número de casos de teste (nct)	Associações não executáveis (nex)
nct = -13.8 + 4.35 t	nex = -33.3 + 7.79 t
nct = -14.9 + 4.06 t + 0.454 variav	nex = -27.5 + 4.08 t + 2.72 variav
nct = -13.5 + 3.72 t + 0.288 def	nex = -20.3 + 1.93 t + 1.82 def
nct = -13.5 + 3.72 t + 0.534 nodef	nex = -19.1 + 1.05 t + 3.94 nodef
nct = 7.21 + 0.0757 potducam	nex = 2.77 + 0.160 potducam

Critério todos potenciais-du -caminhos (CR3)

Número de casos de teste (nct)	Caminhos não executáveis (nex)
nct = -16.3 + 4.86 t	nex = -152 + 31.6 t
nct = -17.4 + 4.58 t + 0.454 variav	nex = -173 + 26.7 t + 8.37 variav
nct = -16.4 + 4.23 t + 0.333 def	nex = -154 + 23.6 t + 4.33 def
nct = -16.5 + 4.24 t + 0.624 nodef	nex = -153 + 23.9 t + 7.66 nodef
nct = 7.04 + 0.0976 potducam	nex = -14.3 + 0.766 potducam

Tabela 6.5: Síntese dos Modelos Excluídas as Unidades Recursivas e a Unidade UNROTATE

Modelo	Expr.	R2	S
M1 (t-ctv)	ct = 2.92 + 1.72 t	25.0	10.06
M2 (t-ctv)	ct = 2.80 + 1.67 t + 0.064 variav	25.1	10.27
M3 (t-ctv)	ct = 2.29 + 1.24 t + 0.299 def	29.0	10.00
M4 (t-ctv)	ct = 1.50 + 0.885 t + 1.02 nodef	32.6	9.743
M5 (t-ctv1)	ct = - 17.9 + 5.18 t	81.1	5.389
M6 (t-ctv1)	ct = - 18.9 + 4.94 t + 0.397 variav	82.1	5.383
M7 (t-ctv1)	ct = - 18.0 + 4.60 t + 0.315 def	83.7	5.123
M8 (t-ctv1)	ct = - 18.0 + 4.50 t + 0.669 nodef	83.8	5.113
M9 (t-ctv11)	ct = - 16.3 + 4.86 t	82.2	4.837
M10 (t-ctv11)	ct = - 17.4 + 4.58 t + 0.454 variav	83.7	4.750
M11 (t-ctv11)	ct = - 16.4 + 4.23 t + 0.333 def	85.8	4.433
M12 (t-ctv11)	ct = - 16.5 + 4.24 t + 0.624 nodef	85.0	4.545
M13 (t-ctv12)	ct = - 16.0 + 4.86 t	72.7	5.716
M14 (t-ctv12)	ct = - 16.8 + 4.53 t + 0.454 variav	74.5	5.824
M15 (t-ctv12)	ct = - 16.1 + 4.24 t + 0.320 def	76.7	5.600
M16 (t-ctv12)	ct = - 16.0 + 4.15 t + 0.678 nodef	76.8	5.577
M17 (t-ctv13)	ct = - 12.9 + 4.26 t	72.2	4.639
M18 (t-ctv13)	ct = - 13.7 + 3.81 t + 0.560 variav	76.2	4.412
M19 (t-ctv13)	ct = - 12.8 + 3.57 t + 0.344 def	79.1	4.133
M20 (t-ctv13)	ct = - 13.0 + 3.62 t + 0.633 nodef	77.5	4.290

Modelo	Expr.	C12	R2	S
MD1 (t-ctv)	ct = 7.12 + 0.0968 ducam	10.51	82.1	4.909
MD2 (t-ctv1)	ct = 7.04 + 0.0976 ducam	10.03	82.7	5.155
MD3 (t-ctv11)	ct = 6.93 + 0.100 ducam	8.65	78.9	5.257
MD4 (t-ctv12)	ct = 7.44 + 0.0879 ducam	8.55	78.5	5.031
MD5 (t-ctv13)	ct = 7.55 + 0.0854 ducam	6.46	68.7	4.92

Tabela 6.6: Síntese de Parâmetros dos Modelos da Tabela 6.5

Modelo	C1	C2	C3	C4	t	Num.Pontos
M1	2.83				3-15	26
M2	2.45	0.15			3-15	26
M3	1.70		1.14		3-15	26
M4	1.13			1.61	3-15	26
M5	9.50				3-12	23
M6	8.38	1.02			3-12	23
M7	7.52		1.80		3-12	23
M8	7.08			1.82	3-12	23
M9	9.60				3-12	22
M10	8.49	1.32			3-12	22
M11	7.77		2.20		3-12	22
M12	7.38			1.91	3-12	22
M13	7.29				3-10	22
M14	6.30	1.16			3-10	22
M15	5.95		1.82		3-10	22
M16	5.65			1.84	3-10	22
M17	7.02				3-10	21
M18	6.05	1.73			3-10	21
M19	5.88		2.44		3-10	21
M20	5.65			2.05	3-10	21

Capítulo 7

Conclusões e Trabalhos Futuros

Na primeira seção deste capítulo são apresentadas as conclusões resultantes das atividades desenvolvidas nesta tese; na Seção 7.2, são discutidas as principais contribuições da tese; finalmente, na última seção são propostos e discutidos os trabalhos de pesquisa futuros.

7.1 Conclusões

Nesta tese investigou-se, essencialmente, o uso de informações de fluxo de dados para derivar critérios de teste estrutural de software. Este é um enfoque promissor, pois uma interpretação possível do teste baseado em fluxo de dados é a de que comandos que têm uma relação de fluxo de dados são provavelmente partes de uma mesma função, e devem ser exercitados (testados) juntamente, pelo menos uma vez; de certa forma, esta abordagem fornece uma conexão com a especificação e com outras informações do produto em teste [H0W87, URA88].

Resultante destes estudos, foram introduzidas a Família de Critérios Potenciais Usos e a correspondente família de critérios executáveis, obtida pela eliminação dos caminhos e associações não executáveis. Uma característica dos critérios Potenciais Usos, que os distingüe dos demais critérios de teste estrutural baseado em fluxo de dados, é que as associações são estabelecidas e requeridas, independentemente da ocorrência explícita de um uso de variável; todos os demais critérios baseados em fluxo de dados exigem a ocorrência de um uso para que um elemento seja requerido. Essa característica foi obtida com a introdução do conceito *potencial uso* que culminou na definição de *potencial-du-caminho*. Os potenciais-du-caminhos a partir de um nó i, de uma certa forma, refletem os possíveis efeitos da mudança de estado do programa, decorrentes das definições de variáveis que ocorrem no nó i, nos processamentos subsequentes; o subgrafo *grafo(i)* associado ao nó i, agrega todos esses possíveis efeitos.

As propriedades teóricas dos Critérios Potenciais Usos foram analisadas utilizando-

se a relação de inclusão e o conceito de complexidade, definidos no Capítulo 2; esses dois aspectos: a complexidade e a relação de inclusão, refletem as propriedades mínimas que devem ser preenchidas por um critério de teste C, a saber:

i) incluir o critério *todos os ramos*, ou seja, um conjunto de casos de teste que exercite os elementos requeridos pelo critério C deve exercitar todos os ramos do programa;

ii) requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; isto equivale ao critério C incluir o critério *todas-def*; e

iii) requerer um conjunto de casos de teste finito.

A principal desvantagem dos critérios baseados em análise de fluxo de dados é que na presença de caminhos não executáveis, eles não garantem a inclusão do critério todos os ramos. Segundo Frankl, os critérios efetivamente utilizados na prática são os critérios de fluxo de dados executáveis; são esses critérios que captam os aspectos semânticos do programa. Os critérios Potenciais Usos estabelecem uma hierarquia de critérios entre os critérios todos os ramos e todos os caminhos, mesmo na presença de caminhos não executáveis. Ainda, são mais adequados para auxiliar na identificação de falhas causadas por dependências de fluxo de dados ausentes, originadas por usos ausentes de variáveis. Deve-se observar que nenhum outro critério de teste estrutural baseado em fluxo de dados inclui os critérios Potenciais Usos.

Na mesma linha de argumentação de Weyuker, para considerar *a inclusão de todos os comandos executáveis* como um dos axiomas entre os axiomas que deveriam ser satisfeitos por um bom critério de teste, pode-se propor que a *inclusão do critério todas-definições* seja estabelecido como mais um dos axiomas que devem ser satisfeitos por um bom critério de teste.

Um fator que contribui para aumentar o custo das atividades de teste é o número de casos de teste necessários para satisfazer o critério selecionado, ou seja, para exercitar todos os elementos requeridos. Mostrou-se que os Critérios Potenciais Usos têm complexidade de ordem exponencial, assim como todos os demais critérios baseados em fluxo de dados; esses resultados retificam resultados publicados por outros pesquisadores da área. Esses fatos ressaltam a importância dos estudos empíricos conduzidos para avaliar o custo da aplicação desses critérios, e a importância da disponibilidade de uma ferramenta de apoio.

Para a definição de uma ferramenta de teste, denominada POKE-TOOL, considerou-se que era extremamente importante a facilidade de incorporar novos critérios de teste, viabilizando o estudo comparativo, teórico e empírico, entre eles, uma vez que, cada um dos critérios introduzidos enfocam características e classes distintas de erros. Por exemplo, o critério *todos caminhos OI simples* objetiva captar os efeitos das entradas nas saídas dos programas, o que pode ser útil, pois força um melhor entendimento do programa e a contraposição deste com sua especificação.

Com esse propósito em mente, quatro modelos básicos de implementação de ferramentas de teste estrutural foram caracterizados: *Modelo de Grafo de Fluxo de Controle; Modelo de Instrumentação; Modelo de Dados; e, Modelo de Descrição dos Elementos Requeridos.* Investigou-se o uso do conceito de arco essencial (arco primitivo), introduzido por Chusho [CHU87], no contexto de teste estrutural baseado em fluxo de dados; foram propostas modificações no algoritmo inicial de Chusho para a obtenção do grafo com redução de herdeiros; esse algoritmo modificado foi denominado *Algoritmo de Redução de Herdeiros de Fluxo de ados.*

A ferramenta POKE-TOOL, proposta nesse trabalho, possui as seguintes características: é uma ferramenta de suporte à aplicação de um conjunto de critérios fazendo, basicamente, análise de cobertura para um conjunto de casos de teste. Ela fornece uma interface simples para o usuário, mas fornece uma organização dos dados de teste que só é encontrada nas ferramentas comerciais. Leva em conta considerações inter-procedurais, que tornam o teste com a POKE-TOOL mais exaustivo e abrangente do que o teste com a maioria das ferramentas; e permite que o usuário a configure para novas linguagens, que é uma característica muito interessante dada a enorme gama de linguagens em uso hoje em dia.

Utilizou-se a POKE-TOOL na aplicação de um benchmark com o objetivo de avaliar empiricamente os critérios Potenciais Usos. Com a aplicação do benchmark, obtiveram-se resultados bastante interessantes. Em geral, pode-se dizer que esses critérios, do ponto de vista prático, são factíveis e demandam um número de casos de teste relativamente pequeno. Vários modelos, bastante razoáveis, para estimar as variáveis resposta: número de casos de teste requeridos (*nct*) e número de caminhos não executáveis (*nex*), foram obtidos. Identificaram-se modelos adequados para diversas fases de desenvolvimento de software. Observou-se ainda que o *conceito potenciais-du-caminhos* é uma métrica de complexidade robusta e que reflete características importantes do software para as atividades de teste, depuração e manutenção. Com base nessa métrica, duas medidas disponíveis na POKE-TOOL foram discutidas: Cobertura Total e Cobertura de Fluxo de Dados.

Apontou-se que os conceitos básicos que norteiam a definição de um critério de teste são extremamente importantes e que, na realidade, um critério de teste pode ser visto como uma medida de complexidade. Neste sentido, os estudos comparativos e propriedades de critérios de teste podem ser aplicados às medidas de complexidade.

Modelos para estimar o esforço das atividades de teste são iniciativas que contribuem e motivam uma prática mais sistemática e disciplinada no desenvolvimento de um produto de software. A quantificação do esforço de teste pode então ser incorporada em planos de teste e auxiliar nas atividades de garantia de qualidade. Esta prática, em geral, leva a um aprimoramento da qualidade global do produto e o custo das atividades de outras fases do desenvolvimento pode ser sensivelmente reduzido.

7.2 Contribuição da Tese

A contribuição da tese é caracterizada por diversos resultados pontuais referentes aos diversos tópicos abordados nesta tese. Os mais relevantes são sintetizados e discutidos a seguir.

- *Introdução dos Critérios Potenciais Usos.* A introdução desses critérios e do conceito *potencial uso*, é uma contribuição relevante da tese, uma vez que, fundamentalmente, é este conceito que garante a inclusão do critério todos os ramos mesmo na presença de caminhos não executáveis, para classes de programas que satisfazem a propriedade LDEN — pelo menos uma definição no nó de entrada. Os critérios Potenciais Usos requerem associações independentemente da ocorrência explícita de uma referência a uma determinada definição; se um uso dessa definição pode existir — *um potencial uso* — a potencial associação é requerida.

- *Determinação de uma estrutura de fluxo de controle que maximiza a geração de potenciais-du-caminhos.* A obtenção dessa estrutura de controle permitiu que erros de análise de complexidade de outros critérios de fluxos de dados fossem identificados; por exemplo, na análise de complexidade dos critérios DFCF e do critério todos caminhos OI simples. Esses resultados errados têm sido utilizados e referenciados em diversos trabalhos de pesquisa: [NTA88, URA88, WEY90], por exemplo. Mostrou-se que todos os critérios baseados em análise de fluxo de dados têm complexidade maior do que 2^t. Outro ponto importante foi identificar que a presença de caminhos não executáveis, influencia não somente a ordem parcial, mas também a análise de complexidade desses critérios.

- *POKE-TOOL: Determinação de Modelos Básicos para Automatização de Critérios de Teste Estrutural.* A relevância de uma ferramenta, que apoie a aplicação de métodos e critérios de teste, é nítida. A proposição e especificação da ferramenta de teste POKE-TOOL constitui uma contribuição desta tese. Sintetizaram-se quatro modelos básicos que abrangem tópicos comuns à automatização de qualquer critério de teste estrutural: *modelo de descrição dos elementos requeridos*; *modelo de fluxo de controle*; *modelo de instrumentação*; e *modelo de fluxo de dados*. Um dos objetivos foi caracterizar um conjunto de recursos que pudesse facilitar a incorporação de outros critérios de teste na ferramenta de teste. Esses modelos são fundamentais para uniformidade e consistência nos estudos de comparação entre esses critérios. Uma característica interessante introduzida no modelo de fluxo de dados é a análise de pior caso realizada para argumentos de chamadas de subprogramas (análise interprocedural); além de ser mais abrangente quanto aos testes requeridos, essa característica facilita a realização do teste de integração.

- *Uso do Conceito de Arco Essencial no Contexto de Fluxo de Dados.* Na definição dos modelos básicos, considerou-se o uso do conceito de arco primitivo

no contexto de fluxo de dados e propôs-se uma variante do algoritmo introduzido por Chusho para obtenção do grafo reduzido de herdeiros; esse algoritmo modificado foi denominado *Algoritmo de Redução de Herdeiros de Fluxo de Dados*. Nesse sentido introduziu-se o conceito de *arco primitivo de fluxo de dados*. A partir desses conceitos e do grafo def foi introduzido o conceito de *grafo(i)*, estabelecendo-se uma base de descrição dos elementos requeridos. A redução do número de associações requeridas, e consequentemente, a serem monitoradas, é bastante significativa; isto implica, claramente, numa redução dos custos globais da atividade de teste.

- *Aplicação e análise do benchmark para a avaliação empírica dos critérios Potenciais Usos*. Uma contribuição importante foi evidenciar que o uso de informações sobre as variáveis do programa são relevantes na determinação de modelos para estimar as variáveis respostas: número de casos de teste requeridos (*nct*) e número de caminhos não executáveis (*nex*); essas informações devem, portanto, ser consideradas em estudos posteriores semelhantes ao apresentado neste trabalho. Um outro resultado foi ressaltar que o *número de potenciais-du-caminhos* é uma medida de complexidade robusta e que reflete características importantes do software para as atividades de teste, depuração e manutenção. Observou-se, por exemplo, que o número de potenciais-du-caminhos tem uma correlação bastante alta com o número de caminhos não executáveis e com o número de casos de teste necessários para satisfazer os critérios Potenciais Usos, constituindo um indicativo bastante forte da dificuldade de condução das atividades de teste. Com base nessa métrica, duas medidas de cobertura, disponíveis na POKE-TOOL, foram propostas e discutidas: Cobertura Total e Cobertura de Fluxo de Dados.

Finalizando, considera-se que os objetivos iniciais foram atingidos, podendo-se caracterizar contribuições tanto no plano teórico, de análise de propriedades e características dos critérios de teste, que usam informações de fluxo de dados para derivar os requisitos de teste, como no plano prático, com a aplicação e análise do benchmark.

Obviamente, os resultados obtidos constituem apenas um embrião de um conjunto de outras atividades de pesquisa a serem desenvolvidas; porém, espera-se, que seja um embrião promissor e motivador. Sugestões e direções de pesquisa e alguns trabalhos que estão sendo conduzidos são discutidos na seção seguinte.

7.3 Trabalhos Futuros

Nos diversos tópicos abordados nesta tese podem-se visualizar direções de pesquisa futura relevantes para a área de Teste de Software e para a área de Engenharia de Software de uma maneira geral.

Uma dessas direções consiste em definir um ambiente para o suporte automatizado às atividades de teste, depuração e manutenção baseado em análise de fluxo de dados. O apoio, por exemplo, ao teste de regressão baseado em análise de fluxo de dados tem sido alvo de pesquisa em diversos centros, por exemplo [OST88]. Nesse contexto, estão sendo conduzidas atividades de pesquisa para o apoio ao teste de regressão baseado em análise de fluxo de dados. Observe-se que o apoio ao teste de regressão envolve o apoio à manutenção das informações de teste propriamente ditas, como por exemplo, a manutenção do conjunto de casos de teste.

Considerando que cada um dos critérios de fluxo de dados enfocam características e classes distintas de erros, a combinação desses conceitos, objetivando o estabelecimento de um critério mais rigoroso e que assimile os pontos importantes dos critérios correspondentes, é uma atividade que está sendo conduzida. Tomando-se como ponto central o conceito potencial uso, o conceito de concatenação de associações e conceitos pertinentes ao teste de laço, está sendo definida uma hierarquia de classes de critérios de teste de programas baseados em fluxo de dados, onde a complexidade da K-ésima classe é menor ou igual ao número de caminhos com k iterações de cada laço.

Um ponto relevante é determinar a influência das diversas abordagens adotadas para o estabelecimento de Modelo de Dados. Exemplificando, o enfoque adotado para variáveis compostas não é conservador, pois alguns fluxos de dados existentes podem não ser requeridos. Um enfoque conservador, semelhante ao adotado para as variáveis definidas por referência, poderia ter sido adotado para variáveis compostas. Pretende-se avaliar a influência dessas abordagens na capacidade de revelar erros, assim como no custo da aplicação dos critérios.

Uma vez que a presença de caminhos não executáveis altera significativamente as propriedades dos critérios, um estudo mais abrangente da ordem parcial entre os critérios de teste estrutural e a análise da complexidade desses critérios na presença de caminhos não executáveis, constituem uma direção de pesquisa bastante importante. Essa análise, de certa forma, está sendo conduzida juntamente com os aspectos de combinação dos critérios de teste.

Um outro tópico que será alvo de pesquisa e desenvolvimento é avaliar a utilização desses conceitos em outros níveis de teste, por exemplo: o teste de integração. Essa linha de pesquisa envolve atividades de abstrair e combinar esses conceitos com conceitos da técnica funcional de teste, uma vez que esta técnica é mais adequada, em geral, para o teste de integração.

A importância de caminhos não executáveis nas atividades de teste ficou bastante

clara ao longo da exposicão desta tese. Estudos para incorporar na POKE-TOOL facilidades para tratamento de caminhos não executáveis e estudos de comparação entre métricas e heurísticas para previsão e determinação de caminhos não executáveis estão sendo conduzidos [VER91a, VER91b]. Esses estudos contribuirão para a definição de estratégias de geração de casos de teste baseadas em fluxos de dados.

Uma vez que as hipóteses consideradas no início do experimento (benchmark) foram comprovadas, ou pelo menos existem fortes evidências nesta direção, este experimento está sendo repetido utilizando-se critérios funcionais para a elaboração dos conjuntos iniciais de casos de teste; um dos objetivos é analisar o grau de cobertura desses critérios em relação aos critérios Potenciais Usos. Outras áreas de aplicação serão alvo de estudos semelhantes: por exemplo, análise numérica, estatística, otimização, etc. Com um volume maior de dados caracterizar-se-ão modelos de estimativas gerais, para áreas específicas e para classes de estrutura de controle, por exemplo. Uma classe de estrutura de interesse seria a composta por estruturas similares à estrutura que maximiza o número de potenciais-du-caminhos; essa classe determinaria o que poderíamos definir de *Modelo de Estimativa Conservadora.* Outra classe seria a definida por estruturas de controle que minimizam a geração de potenciais-du-caminhos, como por exemplo, estruturas de controle que contêm predominantemente laços, e definiria o que poderíamos chamar de *Modelo de Estimativa Mínima.*

Tanto do ponto de vista teórico como empírico, um dos objetivos é avaliar a adequabilidade desses critérios a classes e tipos de erros; a classificação dos tipos de erros e caracterização de benchmarks, preferencialmente com erros conhecidos, são atividades indispensáveis e de grande contribuição a esses estudos. Os estudos teóricos teriam por objetivo, por exemplo, caracterizar classes de erros que garantidamente seriam revelados pelos diversos critérios.

Mais especificamente no contexto da POKE-TOOL, a incorporação de facilidades de visualização, de manutenção da sessão de trabalho, de processamento batch, de definição de "drivers" e "stubs", entre outras, é um dos objetivos futuros. A configuração dessa ferramenta para as linguagens Pascal, COBOL e FORTRAN está em andamento.

Finalmente, um ponto aparentemente específico, mas que pode envolver conceitos teóricos fundamentais, seria minimizar o número de associações requeridas; esta atividade envolveria o estudo de relação de inclusão e de dominância entre os grafos grafo(i), por exemplo. Isto implicaria na redução de custo de teste e de manutenção, uma vez que o tempo associado à análise de cobertura seria minimizado. A avaliação empírica do uso de arcos primitivos neste contexto também será alvo de estudo como continuidade desta tese.

Referências:

[**ACH90**] Achcar, J.A. e Rodrigues, J., *Introdução de Estatística para Ciências e Tecnologia* , ICMSC/USP — São Carlos, SP, Brasil, 1990.

[**AND86**] Andriole, S.J. (editor), *Software Validation, Verification, Testing and Documentation* , Petrocelli Books, New Jersey, 1986.

[**BEI83**] Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold Company, New York, USA, 1983.

[**BEI84**] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand Reinhold Company, New York, USA, 1984.

[**BEI90**] Beizer, B., Correspondência Pessoal, 1990.

[**BIE89**] Bieman, J. e Schultz,J., "Estimating the Number of Test Cases Required to Satisfy The All-du-paths Testing Criterion", in *Proc. of the ACM SIGSOFT'89 - Third Symposium on Software Testing, Analysis and Verification (TAV3)*, Flórida, USA, Dez, 1990.

[**CAR91**] Carnassale, M., "GFC — Uma Ferramenta Multilinguagem para Geração de Grafo de Programa", *Tese de Mestrado*, DCA/FEE/UNICAMP Fev, 1991.

[**CLA82**] Clark, L.A., Hassel, J., Richardson, D.J. (1982) "A Close Look at Domain Testing", *IEEE Trans. on Software Eng.*, Vol. SE - 8, No. 4, Abr. 1982, pp.380-390.

[**CLA86**] Clarke, L., Podgurski,A., Richardson, D.J. and Zeil, S.J., "A Comparison of Data Flow Path Selection Criteria", in *Proc. of the 8th Int'l Conf. on Software Engineering*, Ago. 1985, pp. 244-251.

[**CHA89**] Chaim, M.L., Maldonado, J.C. e Jino, M., "Modelando a Determinação de Potenciais Du-Caminhos Através da Analíse de Fluxo de Dados," in *Proc. III Simp. Bras. Eng. de Software*, Recife, P.E., Out. 1989.

[**CHA91a**] Chaim, M.L., Maldonado, J.C. and Jino, M., *Projeto / Implementação de uma Ferramenta de Teste de Software, Relatório Técnico* DCA/RT/007/91 - DCA/FEE/UNICAMP - 1991.

[**CHA91b**] Chaim, M.L., Maldonado, J.C. e Jino, M., *Manual de Configuração da POKE-TOOL, Relatório Técnico* DCA/RT/008/91 - DCA/FEE/UNICAMP - 1991.

[**CHA91c**] Chaim, M.L., Maldonado, J.C. e Jino, M., *Manual do Usuário da POKE-TOOL,Relatório Técnico* DCA/RT/009/91 - DCA/FEE/UNICAMP - 1991.

[**CHA91d**] Chaim, M.L., "POKE-TOOL — Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados", *Tese de Mestrado*, DCA/FEE/UNICAMP - Campinas, SP, Brasil, Abril 1991.

[**CHA91e**] Chaim, M.L., Maldonado, J.C., Jino, M., "POKE-TOOL — Uma Ferramenta para Suporte à Aplicação dos Critérios Potenciais Usos para Teste de Programas", 1991 (*submetido para publicação*).

[**CHU87**] Chusho, T., "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing", *IEEE Trans. Software Eng.* Vol. SE-13, No.5, Maio 1987, pp. 509-517.

[**COW88**] Coward P.D., "A Review of Software Testing", *Information and Software Technology*, vol.30, No.3. Abr 1988, pp.189-198.

[**DYE90**] Dyer,M. e Kouchakdjian, "Correctness Verification: Alternative to Structural Software Testing", *IEEE Software*, Vol.32, No.2, Jan./Fev., 1990.

[**DEN82**] Dennis, C.R. e Sanford, W., *Residuals and Influence in Regression*, Chapman and Hall, New York, USA, 1982.

[**DEU82**] Deutsch, M. S. *Software Verification and Validation.*, Englewood Cliffs, Prentice-Hall, 1982.

[**DUR84**] Duran, J. and Ntafos, S. "An Evaluation of Random Testing" - *IEEE Trans. on Software Eng.*, Vol. SE - 10, No. 7, Jul. 1984, pp.438-444.

[**FRA85**] Frankl, F.G. and Weyuker, E.J., "Data Flow Testing Tool", in *Proc. Softfair II*, San Francisco, CA, Dez. 1985, pp.46-53.

[**FRA86**] Frankl, F.G. and Weyuker, E.J., "Data Flow Testing in the Presence of Unexecutable Paths", in *Proc. Workshop on Software Testing*, Banff, Canada, Jul. 1986, pp. 4-13.

[**FRA87**] Frankl, F.G., "The Use of Data Flow Information for the Selection and Evaluation of Software Test Data", *Ph.D dissertation*, New York Univ., New York, Out. 1987.

[**FRA88**] Frankl, F.G. and Weyuker, E.J., "An Applicable Family of Data Flow Testing Criteria", *IEEE Trans. on Software Eng.*, Vol. 14, No. 10, Oct. 1988, pp. 1483-1498.

[**GEL88**] Gelperin D. e Hetzel, B., "The Growth of Software Testing", *Comm. of ACM*, Vol.31, No.06, Jun. 1988, pp. 687-695.

[**GHE87**] Ghezzi, C. e Jazayeri, M., *Programming Languages Concepts*, John Wiley and Sons, segunda edição, New York, 1987.

[**GOD75**] Goodenough, J. e Gerhart, S.L. "Toward a Theory of Test Data Selection", *IEEE Trans. on Software Eng.*, Vol. SE - 1, 1975.

[**GOU83**] Gourlay, J.S., "A Mathematical Framework for the Investigation of Testing", *IEEE Trans. on Software Eng.*, Vol. SE - 9, No. 6, Nov.. 1983, pp.680-709.

[**DEM78**] De Millo, R.A., Lipton, R.J. e Sayward, F.G., "Hints on Test Data Selection for the Practicing Programmer", *Computer*, Abr., 1978.

[**HAL91**] Hall, P.A.V., "Relationship between Specifications and Testing", *Information and Software Technology*, Vol. 33, No.1, Jan/Fev, 1991, pp.47-52.

[**HAM88**] Hamlet, R. and Taylor, R. "Partition Testing does not Inspire Confidence" in *Proc. Second Workshop on Software Testing, Verification and Analysis*, Banff, Canada, Jul., 1988, pp.206-215.

[**HEC77**] Hecht, M.S., *Flow Analysis of Computer Programs.*, North Holland, New York, 1977.

[**HER76**] Herman, P.M., "A Data Flow Analysis Approach to Program Testing", *The Australian Computer Journal*, Vol. 8, No.3, Nov.1976, pp.92-96.

[**HOL79**] Holthouse, M.A., Hatih, M.J., "Experience with Automated Testing Analysis", *IEEE Computer*, Ago. 1979, pp.33-36.

[**HOW75**] Howden, W.E., "Methodology for the Generation of Program Test Data", *IEEE Transactions on Computer*, C-24(5), 1975, pp.554-559.

[**HOW78**] Howden, W.E., "An evaluation of the Effectiveness of Symbolic Testing", *Software Practice and Experience*, 8, 1978.

[**HOW81**] Howden, W.E., "Completeness Criteria for Testing Elementary Program Functions". in *Proc. Fifth International Conference on Software Engineering* , San Diego, IEEE, 1981.

[**HOW87**] Howden W.E., *Functional Program Testing and Analysis*, McGraw-Hill, USA, 1987.

[**HOR87**] Horwitz, S., Demers, A. and Teitelbaum, T., "An Efficient General Iterative Algorithm for Dataflow Analysis," *Acta Informatica* , vol. 24, 1987. pp. 679-694.

[**JEN89**] Jeng, B. and Weyuker, E.J. "Some Observations on Partition Testing", in *Proc. of the ACM SIGSOFT'89 - Third Symp. on Software Testing, Analysis and Verification (TAV3)*, Flórida, USA, Dez. 1990, pp.38-47.

[**JONES86**] Jones, C.B., *Systematic Software Development Using VDM*, Prentice-Hall International, Great Britain, 1986.

[**JON90**] Jones, G.W., *Software Engineering*, John Wiley & Sons, USA, 1990.

[**KAM77**] Kam, J.B. and Ullman, J.D., "Monotone Data Flow Frameworks," *Acta Informatica* , vol. 7, 1977, pp. 305-317.

[**KER78**] Kernighan, B.W. e Ritchie, D.M., *The C Programming Language.*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[**KER81**] Kernighan, B.W. e Plauger, P.J., *Software Tools in Pascal*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[**KIN76**] King, J.C., "Symbolic Executation and Program Testing", *Comm. of ACM*, Vol. 19, No.7, 1976, pp.385-394.

[**KOR85**] Korel, B. and Laski, J., "A Tool for Data Flow Oriented Program Testing", in *Proc. Softfair II*, San Francisco, CA, Dez. 1985, pp. 34-38.

[**KOR87**] Korel, B., "The Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol. 24, No. 2, Jan. 1987, pp. 103-108.

[**LAS83**] Laski, J.W. and Korel, B., "A Data Flow Oriented Program Testing Strategy", *IEEE Trans. Software. Eng.*, Vol. SE-9, No. 3, May 1983 pp. 347-354.

[**LUT90**] Lutz, M., "Testing Tool," *IEEE Software* , Vol. 7, No. 3, Maio 1990, pp. 57.

[**MAY90**] Mayhauser, A., *Software Engineering - Methods and Management*, Academic Press Inc, USA, 1990.

[**MAL88a**] Maldonado, J.C., Chaim, M.L., Jino, M., "Seleção de Casos de Testes baseada nos Critérios Potenciais Usos", in *Proc. II Simpósio Brasileiro de Engenharia de Software*, Canela, RS, Brasil, Out. 1988, pp. 24-35.

[**MAL88b**] Maldonado, J.C., Chaim, M.L., Jino, M., "Resultados do Estudo de uma Família de Critérios de Teste de Programas baseada em Fluxo de Dados", *Relatório Técnico* - DCA/FEE/UNICAMP - RT/DCA-001/88 - Campinas, SP, Brasil, 1988.

[**MAL89a**] Maldonado, J.C., Chaim, M.L., Jino, M., "Arquitetura de uma Ferramenta de Teste de Apoio aos Critérios Potenciais Usos", in *Proc. XXII Congresso Nacional de Informática* São Paulo, SP, Brasil, Set. 1989.

[**MAL89b**] Maldonado, J.C., Chaim, M.L., Jino, M., "Feasible Potential Uses Criteria Analysis" *Relatório Técnico* - DCA/FEE/UNICAMP - RT/DCA-001/89 - Campinas, SP, Brasil, 1989.

[**MAL91a**] Maldonado, J.C., Chaim,M.L., Jino, M., "Potential Uses Criteria Complexity Analysis", *Notas Técnicas* - DCA/FEE/UNICAMP - TN/DCA-001/91 - Campinas, SP, Brasil, 1991.

[**MAL91b**] Maldonado,J.C., Chaim, M.L., Jino, M., "Using the Essential Branch Concept to Support Data-Flow Criteria Application", 1991 (*submetido para publicação*).

[**MAL91c**] Maldonado, J.C., Chaim, M.L., Jino, M., "Potential Uses Testing Criteria: A Step towards Bridging the Gap in the Presence of Unfeasible Paths", 1991 (*submetido para publicação*).

[**MAL91d**] Maldonado, J.C., Vergĺio, S.R., Chaim, M.L.; Jino, M., "Critérios Potenciais Usos: Análise da Aplicação de um Benchmark ", 1991 (*submetido para publicação*).

[**MAL91e**] J. C. Maldonado, M. L. Chaim, S. R. Vergílio, M.Jino, "Critérios Potenciais Usos: Uma Contribuição para a Atividade de Garantia de Qualidade de Software", in *Proc. Workshop em Avaliação de Qualidade de Software*, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, Maio, 1991.

[**MALE90**] Malevris, N. and Yates, D.F. and Veevers, A., "Predictive Metric for Likely Feasibility of Programs Paths", *Information and Software Technology*, Vol. 32, No. 2, Mar. 1990, pp. 115-118.

[**McC76**] McCabe, T.J., "A Complexity Measure", *IEEE Trans. Software Eng.*, Vol. SE-2, Dez. 1976.

[**MYE79**] Myers, G.J., *The Art of Software Testing.*, Wiley, New York, 1979.

[**NTA84**] Ntafos, S.C., "On Required Element Testing", *IEEE Trans. Software Eng.*, Vol. SE - 10, Nov. 1984, pp. 795-803.

[**NTA88**] Ntafos, S.C., "A Comparison of Some Structural Testing Strategies", *IEEE Trans. Software Eng.*, Vol. 14, No. 6, Jun. 1988, pp. 868-873.

[**OST79**] Ostrand, T.J., Weyuker, E.J., "Error-Based Program Testing", Presented at *1979 Conference on Information Sciences and Systems*, Baltimore, Mar., 1979.

[**OST88**] Ostrand, T.J., Weyuker, E.J., "Using Data Flow Analysis for Regression Testing", *Sixth Annual Pacific Northwest Software Quality Conference*, Portland, Oregon, Set, 1988.

[**POD90**] Podgurski, A. and Clarke, L.A., "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance", *IEEE Trans. on Software Eng.*, Vol. SE - 16, No. 9, Set., 1990, pp. 965-979.

[**PRE87**] Pressman, R.B., *Software Engineering: a practitioner's approach* , segunda edição, MacGraw-Hill, New York, 1987.

[**PRI87**] Price, A.M, Garcia, F. e Purper, C.B., "Visualizando o Fluxo de Controle de Programas", in *Proc. I Simp. Bras. Eng. de Software*, Petropólis, R.J., Out. 1987, pp. 1-9.

[**PRI90**] Price, A.M. e Zorzo, A., "Visualizando o Fluxo de Controle de Programas", in *Proc. IV Simp. Bras. Eng. de Software*, Águas de São Pedro, S.P., Out.1990.

[**RAP82**] Rapps, S., Weyuker, E.J. "Data Flow Analysis Techniques for Test Data Selection" in *Proc. Int. Conf. Software Eng.*, Tokio, Japão, Set. 1982, pp. 272-278.

[**RAP85**] Rapps, S. and Weyuker, E.J., "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, Vol. SE - 11, Abr. 1985, pp. 367-375.

[**RYA76**] Ryan Jr., T.A. and Joiner, B.L. and Ryan, B.F., *MINITAB — Student Handbook*, Wadsworth Publishing Company, Califórnia, 1976.

[**REI90**] Reisman, S., "Management and Integrated Tools," *IEEE Software*, Vol. 7, No. 3, Maio 1990.

[**RIC81**] Richardson, D.J. e Clarke, L.A., "A Partition Analysis Method to Increase Program Reability", *Fith International Conf. on Software Engineering*, 1981, pp.244-253.

[**RIC85**] Richardson, D.J., "Partition Analysis: a Method Combining Testing and Verification", *IEEE Trans. on Software Eng.*, Vol. SE - 11, No. 10, Out. 1985.

[**SEB89**] Sebesta, R.W., *Concepts of Programming Languages.*, Benjamin Cummings Publishing Company, Inc.., Redwood City, 1989.

[**SET81**] Setzer, V.W. and de Melo, I.S.H., *A Construção de um Compilador*, terceira edição, Editora Campus, R.J., Brasil, 1981.

[**SOM89**] Sommerville, I., *Software Engineering*, terceira edição, Addison-Wesley Publ. Company, 1989.

[**VER91a**] Vergílio, S.R., Maldonado, J.C., Chaim, M.L., Jino, M., "Caminhos Não Executáveis na Automação da Atividades de Teste ", 1991 (*submetido para publicação*).

[**VER91b**] Vergílio, S.R., *Tese de Mestrado*, DCA/FEE/UNICAMP - Campinas, SP, Brasil, 1991 (*em preparação*).

[**VOU90**] Vouk, M.A., "Back to Back Testing", *Information and Software Technology*, Vol. 32, No. 1, Jan. 1990, pp. 34-45.

[**WEY80**] Weyuker, E.J., Ostrand T.J., "Theory of Program Testing and the Application of Revealing Subdomains, *IEEE Trans. on Software Eng.*, Vol. SE - 6, 1980.

[**WEY82**] Weyuker, E.J., "On Testing Non-testable Programs", *the Computer Journal*, vol.25, No.4, 1982, pp.465-470.

[**WEY84**] Weyuker, E.J., "The Complexity of Data Flow Criteria for Test Data Selection", *Information Processing Letters*, Vol. 19, No.2, Aug. 1984, pp. 103-109.

[**WEY86**] Weyuker, E.J., "Axiomatizing Software Test Data Adequacy", *IEEE Trans. on Software Eng.*, Vol. SE - 12, No. 12, Dec. 1986, pp. 1128-1138.

[**WEY88a**] Weyuker, E.J., "The Evaluation of Program-Based Software Test Data Adequacy Criteria", *IEEE Trans. on Software Eng.*, Vol. SE - 16, No. 2, Feb. 1990, pp. 121-128.

[**WEY88b**] Weyuker,E.J., "An Empirical Study of the Complexity of Data Flow Testing", in *Proc. of Second Workshop on Software, Verification and Analysis (TAV2)* , Banff, Canada, Jul. 1988.

[**WEY90**] Weyuker, E.J., "The Cost of Data Flow Testing: An Empirical Study", *IEEE Trans. on Software Eng.*, Vol. SE - 16, No. 2, Feb. 1990, pp. 121-128.

[**WIR76**] Wirth, N., *Algorithms + Data Structures = Programs.* Englewood Cliffs, NJ, Prentice-Hall, 1976.

[**WHI80**] White, L.J. e Cohen, E.I., "A Domain Strategy for Computer Program Testing". *IEEE Trans. on Software Eng.*, Vol. SE - 6, No. 3, 1980, pp. 247-257.

[**WHI85**] White, L.J. and Sahay, P.N., "A Computer System for Generating Test Data Using the Domain Strategy", in *Proc. Softfair II*, San Francisco, CA, Dec. 1985, pp. 38-45.

[**WOO80**] Woodward, M.R., Heddley, D. and Hennel, M.A., "Experience with Path Analysis and Testing of Programs", *IEEE Trans. on Software Eng.*, Vol. SE - 6, Maio 1980, pp.278-286.

[**URA88**] Ural, U. and Yang, B., "A Structural Test Selection Criterion", *Information Processing Letters*, Vol. 28, Jul. 1988 pp.157-163.

[**ZWE89**] Zweben, S.H. and Gourlay, J.S., "On the Adequacy of Weuker's Test Data Adequacy Axioms", *IEEE Trans. on Software Eng.*, Vol. SE - 15, No. 4, Abr. 1989, pp. 496-501.

Apêndice A

A Linguagem Intermediária (LI)

Neste apêndice é apresentada uma síntese da descrição da Linguagem LI [CAR91, CHA91a]. A linguagem intermediária (LI) visa a identificar o fluxo de execução em um programa. Basicamente, A LI tem dois tipos de comandos: comandos *seqüenciais* e comandos de *controle de fluxo*. Os comandos *seqüenciais* da LI indicam os comandos das linguagens procedurais que representam uma declaração de variável ou uma computação (comandos de atribuição ou chamadas de procedimentos) e que, portanto, não alteram o fluxo de execução. Os comandos de *controle de fluxo* da LI são equivalentes aos comandos das linguagens procedurais que causam *seleção*, *seleção múltipla*, *iteração* e *transferência incondicional*.

Uma característica própria da LI é que todos os *átomos* da linguagem são seguidos por números que identificam, respectivamente, o início do átomo no arquivo fonte da unidade em teste (a quantos bytes do começo do arquivo se inicia o átomo), o comprimento do átomo (quantos bytes tem o átomo) e a linha onde está o átomo. Dessa maneira, utilizando a notação de Backus-Naur, um átomo da LI teria a seguinte estrutura:

$$< atm\_li > ::\equiv < atomo >< inicio >< comprimento >< linha >.$$

Onde

$$\begin{aligned} < atomo > ::\equiv\ & \$DCL|\$S|\$IF|\$CASE|\$ROTC|\$ROTD \\ & |\$WHILE|\$FOR|\{|\}|\$C|\$NC|\$REPEAT|\$UNTIL \\ & |\$GOTO|LABEL|\$BREAK|\$CONTINUE|\$RETURN \\ & \$CC|\$ELSE \end{aligned}$$

e

$< inicio > ::\equiv NUM$,
$< comprimento > ::\equiv NUM$ e
$< linha > ::\equiv NUM$.

Os terminais acima representam a seqüência de caracteres indicada pelos próprios terminais; com exceção de $\$S$ que indica a sequência de caracteres $\$Sd^n$ onde d pertence a $\{0, 1, \ldots, 9\}$ e $n > 0$, $\$C$ que indica a seqüência $\$C(d^n)d^n$, $\$NC$ que indica a seqüência $\$NC(d^n)d^n$, NUM que indica a seqüência d^n e $LABEL$ que indica uma seqüência de letras e caracteres sempre começando por um caractere.

A utilidade desses ponteiros dos átomos é possibilitar o acesso ao código fonte associado ao átomo da LI, o que vai ser necessário, por exemplo, para a extensão do grafo de fluxo de controle para a geração do *grafo def* e para a instrumentação da unidade em teste.

Na convenção de Backus-Naur [SEB89] adotada aqui, os terminais serão representados em itálico, os não-terminais entre "<" e ">" e os meta-símbolos sublinhados.

Os comandos *seqüênciais* são os seguintes:

$< dcl > ::\equiv \$DCL < inicio >< comprimento >< linha >$ e
$< s > ::\equiv \$S < inicio >< comprimento >< linha >$.

Onde $< dcl >$ denota uma *declaração de variável* e $< s >$ uma computação; sendo que uma computação pode ser uma atribuição de valor a uma variável (através de uma expressão ou chamada de função), ou somente uma chamada de procedimento.

Os comandos de controle de fluxo são os seguintes:

(1) Seleção:

$$< if > ::\equiv < if\_atm >< cond\_atm >< statement_1 > \mid$$
$$< if\_atm >< cond\_atm >< statement_1 >< else\_atm >< statement_2 >$$

onde

$< if\_atm > ::\equiv \$IF < inicio >< comprimento >< linha >$,
$< cond\_atm > ::\equiv \$C < inicio >< comprimento >< linha >$,
$< else\_atm > ::\equiv \$ELSE < inicio >< comprimento >< linha >$ e

$< statement_i >$ é o não-terminal que denota todos os possíveis comandos da LI, agrupados ou não; $< statement_i >$ será definido formalmente mais adiante. Este comando significa o "if" tradicional das linguagens do estilo ALGOL, ou seja, os "comandos" em $< statement_1 >$ serão executados se $< cond\_atm >$ for verdadeiro e os "comandos" em $< statement_2 >$ serão executados caso contrário.

A solução para a ambigüidade que poderia ser gerada pelo encadeamento de *$IF* e *$ELSE* sem delimitadores de bloco { e } foi tomada inspirada na maioria das linguagens do estilo ALGOL, onde o *$ELSE* é associado com o mais recente *$IF* sem *$ELSE*, salvo o uso explícito de chaves que podem forçar a associação apropriada. Observe-se que *$S1* acima representa o primeiro comando sequencial do programa em LI, ou seja, o número que segue os caracteres "$S" indica a ordem em que aparece o comando sequencial no programa. Os números que aparecem em *$C(1)1* indicam, respectivamente, o número de predicados que possui a condição e a ordem de aparição.

(2) Seleção Múltipla:

$< case > ::\equiv$
$< case\_atm >< case\_cond\_atm >$ { { (< rotc_atm > | < rotd_atm >) { < statement > }}$^+$}
onde
$< case\_atm > ::\equiv \$CASE < inicio >< comprimento >< linha >$,
$< case\_cond\_atm > ::\equiv \$CC < inicio >< comprimento >< linha >$,
$< rotc\_atm > ::\equiv \$ROTC < inicio >< comprimento >< linha >$ e
$< rotd\_atm > ::\equiv \$ROTD < inicio >< comprimento >< linha >$.

O não-terminal $< case\_atm >$ representa o átomo que inicia o comando de seleção múltipla, $< case\_cond\_atm >$ representa a condição do comando e $< rotc\_atm >$ representa os possíveis rótulos para as seqüências de comandos indicadas por $< statement >$. O não-terminal $< rotd\_atm >$ representa o rótulo para a seqüência de comandos a ser executada quando a condição não combina com nenhum rótulo $< rotc\_atm >$.

A semântica do comando acima é equivalente ao comando "switch" da linguagem C, isto é, a execução começa no rótulo que ocorreu a combinação e executa os comandos desse rótulo mais os comandos dos rótulos que o seguem, a menos que seja encontrado um comando do tipo "break", que causa a imediata saída do comando de seleção múltipla, ou um comando de desvio incondicional. Esses comandos serão discutidos mais adiante.

(3) Iteração

A LI fornece comandos para iteração tanto para um número fixo de repetições quanto para um número de repetições que depende de uma condição.

No caso de um número fixo de repetições, tem-se um comando semelhante ao "for" das linguagens do estilo ALGOL. O "for" da LI é definido como

$< for > ::\equiv < for\_atm >< s_1 >< cond\_for\_atm >< s_2 >< statement >$
onde
$< for\_atm > ::\equiv \$FOR < inicio >< comprimento >< linha >$ e
$< cond\_for\_atm > ::\equiv \$C < inicio >< comprimento >< linha >$.

O não-terminal $< for\_atm >$ indica o comando "for" da LI, o não-terminal $< s_1 >$ representa a iniciação das variáveis de controle do "for" através de um comando sequencial, $< cond\_for\_atm >$ representa a condição, $< s_2 >$ representa o comando sequencial que altera as variáveis de controle a cada iteração do "for" e $< statement >$ representa o corpo do comando. O comando "for" da LI é inspirado no comando equivalente da linguagem C, possuindo a mesma semântica.

Os comandos de iteração cujo número de repetições é dirigido por uma condição são também equivalentes aos tradicionais "while" e "repeat-until" das linguagens do estilo ALGOL. O "while" da LI é definido como

$< while > ::\equiv < while\_atm >< cond\_while\_atm >< statement >$
onde
$< while\_atm > ::\equiv \$WHILE < inicio >< comprimento >< linha >$ e
$< cond\_while\_atm > ::\equiv \$C < inicio >< comprimento >< linha >$.

$< while\_atm >$ indica o comando "while", $< cond\_while\_atm >$ a condição e $< statement >$ o corpo do "while". O corpo será executado enquanto a condição em $< cond\_while\_atm >$ permanecer verdadeira.

O "repeat-until" da LI é definido como

$< repeat\_until > ::\equiv$
$< repeat\_atm >< statement >< until\_atm >< cond\_until\_atm >$,
onde
$< repeat\_atm > ::\equiv \$REPEAT < inicio >< comprimento >< linha >$,
$< until\_atm > ::\equiv \$UNTIL < inicio >< comprimento >< linha >$ e
$< cond\_until\_atm > ::\equiv (\$C|\$NC) < inicio >< comprimento >< linha >$.

$< repeat\_atm >$ indica o início do comando "repeat-until", $< statement >$ o corpo do comando, $< until\_atm >$ indica o fim do comando e $< cond\_until\_atm >$ representa a

condição de término. A semântica desse comando da LI é igual ao comando equivalente das linguagens "ALGOL-like", ou seja, o corpo da iteração é executado pelo menos uma vez e o teste da condição é realizado depois da execução do corpo. Note-se que a condição de término é composta por *$C* ou *$NC*, isto ocorre porque, na maioria das linguagens "ALGOL-like", o corpo do "repeat-until" é executado até que uma dada condição se verifique; porém, em algumas linguagens como C, o comando "repeat-until" equivalente funciona de maneira que o corpo é executado enquanto a condição permanece verdadeira. Devido a esse fato, a condição desse comando pode ser *$C*, se for o primeiro caso, e aí estaria indicando um "repeat-until" tradicional, ou *$NC*, se for o segundo caso, e nesta situação teríamos que o laço se repete até a negação da condição.

Os comandos de desvio incondicional provocam a mundança do fluxo de execução em um programa. A LI possui um comando de transferência incondicional irrestrito do tipo "goto" e comandos de transferência incondicional controlada; estes últimos têm sua utilização limitada a algumas situações e seu efeito bem previsível.

O comando "goto" da LI é definido como

$< goto > ::\equiv < goto\_atm >< label\_atm >$
onde
$< goto\_atm > ::\equiv \$GOTO < inicio >< comprimento >< linha >$ e
$< label\_atm > ::\equiv LABEL < inicio >< comprimento >< linha >$.

$< goto\_atm >$ representa o comando "goto" da LI e $< label\_atm >$ representa o rótulo para onde deve ser dirigido o fluxo de execução quando é encontrado o comando "goto".

Existem mais três comandos de desvio incondicional na LI, são eles

$< break > ::\equiv \$BREAK < inicio >< comprimento >< linha >$,
$< continue > ::\equiv \$CONTINUE < inicio >< comprimento >< linha >$ e
$< return > ::\equiv \$RETURN < inicio >< comprimento >< linha >$.

Esses comandos de transferência incondicional foram inspirados nos seus homônimos da linguagem C e, por isso, possuem efeitos idênticos. O "break" causa o fim do comando de iteração ("for", "while" ou "repeat-until") mais próximo que o engloba. Ainda, dentro de um comando "case" da LI, o "break" causa o desvio para o primeiro comando fora do "case". O comando "continue" provoca o desvio para a *próxima iteração* do laço que o engloba. No caso dos comandos "repeat-until" e "while", ao encontrar-se o "continue", o fluxo de execução é desviado para o teste da condição da iteração; no caso do comando "for", o fluxo de execução é desviado para o comando

que altera as variáveis de controle. O comando "return" causa o fim do procedimento que está sendo executado e o retorno para a unidade que o chamou.

Até aqui foram descritos os comandos individuais da LI, entretanto, para concluir a definição da LI, ainda falta definir como se agrupam comandos na LI e como esses comandos são organizados em um programa. O não-terminal < *statement* >, muito utilizado acima, representa um único comando da LI ou um agrupamento deles e é definido como

$$
\begin{aligned}
< statement > ::= \; & \{ \; \{ \; < statement > \; \} \; \} \; | \\
& < dcl > \; | \\
& < s > \; | \\
& < if > \; | \\
& < case > \; | \\
& < for > \; | \\
& < while > \; | \\
& < repeat\_until > \; | \\
& LABEL < statement > \; | \\
& < goto > \; | \\
& < break > \; | \\
& < continue > \; | \\
& < return > .
\end{aligned}
$$

Os programas em LI são definidos da seguinte maneira:

$$< program > ::= \{ \; (\; < dcl > \; | < s > \;) \; \} \; \{ \; < statement > \; \}.$$

Um exemplo da tradução de um programa fonte para a LI é fornecido no Apêndice C, que consiste na tradução do programa ENTAB.C para a linguagem LI.

A associação dos comandos do código fonte para a LI não é sempre direta como poderia ser concluído do exemplo citado acima; considere o trecho de programa em Pascal [GHE87]:

```
type day = (sunday,monday,tuesday,wednesday,thursday,friday,saturday);
var week_day: day;
...
    for week_day := monday to friday do
        ...;
```

seria traduzido para

```
$DCL        51     70     2
$DCL       124     18     3
                ...
$FOR       303      3     7
$S1        307     18     7
$C(1)1       0      0     0
$S2        326      9     7
                ...
```

Observe-se que na condição do comando "for" os ponteiros são iguais a 0, indicando que o átomo da LI não possui correspondência no arquivo fonte. Obviamente, a decisão de como mapear a linguagem da unidade para a LI é do usúario configurador da POKE-TOOL.

Apêndice B

Determinação do Grafo de Fluxo de Controle que Maximiza o Número de Potenciais-du-caminhos

Neste apêndice mostra-se que o grafo de fluxo de controle da Fig. B maximiza o número de du-caminhos, e portanto, o número de potenciais-du-caminhos, dado por

$$((11/2)t + 9) * 2^t - 10 * t - 9$$

. A prova aqui apresentada é uma prova construtiva e assume que os comandos de controle de fluxo de execução têm no máximo duas alternativas; a extensão dessa prova para estrutura de controles que admitem mais de duas alternativas, como por exemplo a estrutura CASE, é praticamente imediata.

Considere um grafo de fluxo de controle $G(N, E, s)$. Um programa P pode ser modelado por G realizando-se as seguintes transformações:

$T1 : B \rightarrow N$ onde

$B = \{B_i \mid B_i$ é um bloco de programa de $P\}$

$$T1(B_i) = \begin{cases} n_i \in N, & \text{se } B_i \neq B_I \\ s \in N, & \text{se } B_i = B_I \text{ onde} \end{cases}$$

$B_I =$ Bloco Inicial e

$\forall B_i, B_j \in B$ se $B_i \neq B_j \Rightarrow T1(B_i) \neq T1(B_j)$.

$T2 : F \subseteq B \times B \rightarrow E$

$$T2(B_i, B_j) = (n_i, n_j) \in E$$

(B_i, B_j) indica que o fluxo de controle pode ser transferido do bloco B_i para o bloco B_j e

$\forall (B_i, B_j), (B_k, B_l) \in F$ se $(B_i, B_j) \neq (B_k, B_l) \Rightarrow T2(B_i, B_j) \neq T2(B_k, B_l)$.

Seja $suc(B_i)$ o conjunto de blocos B_j que sucedem execucionalmente o bloco B_i, isto é, $suc(B_i) = \{B_j \mid T2(B_i, B_j) \in E\}$. Considerando-se apenas comandos de transferência de controle com no máximo duas alternativas, pode-se estabelecer que $0 \leq \mid suc(B_i) \mid \leq 2$. Seja $suc(n_i)$ o conjunto definido pelos nós n_j que sucedem execucionalmente n_i, isto é, $suc(n_i) = \{n_j \mid (n_i, n_j) \in E\}$. Aplicando-se $T1$ e $T2$, pode-se observar que $0 \leq \mid suc(n_i) \mid \leq 2$.

OBS1: Todo programa P escrito utilizando-se apenas estruturas de controle com no máximo duas alternativas pode ser representado por um grafo $G(N, E, s)$ onde $\forall i \in N$, $out - degree(i) \leq 2 (out - degree(i) = \mid suc(i) \mid)$.

A Fig. B é um esquema geral de du-caminhos (potencial-du-caminhos). É fácil observar que:

OBS2: Para maximizar o número de du-caminhos um grafo de fluxo de controle G deve ter no máximo um laço; adicionalmente, laços intermediários devem ser evitados.

A partir da definição de potencial-du-caminho, dado um potencial-du-caminho $p = (i_1, i_2, \ldots, i_{n-1}, i_n)$, todo caminho incluído em p, começando no nó i_1, é um potencial-du-caminho; dado um du-caminho $q = (i_1, i_2, \ldots, i_n)$, todo caminho incluído em q, começando no nó i_1, com alocação adequada de usos das variáveis definidas no nøi, estabelece um du-caminho. Então:

OBS3: Para maximizar o número de du-caminhos, deve-se considerar grafos que explorem o aspecto combinatorial de composição de caminhos e que maximizem o número de nós.

Ainda, as seguintes observações são relevantes:

OBS4: As construç oes básicas ilustradas na Fig. B representam as estruturas de controle IF-THEN-ELSE, WHILE, e REPEAT-UNTIL.

OBS5: A partir do primeiro nó do grafo de fluxo de controle G_t, ilustado na Fig.B, correspondente a uma sequência de t estruturas IF-THEN-ELSE, existem $4 * (2^t - 1)$ du-caminhos, no pior caso; e existem $4 * (2^t - 1)$ caminhos que incluem o último nó.

OBS6: G_t tem $2^{t+4} - 10 * t - 16$ du-caminhos, no pior caso.

OBS7: O grafo de fluxo de controle da Fig.B, tem $2^{t+4} - 10 * t - 12$ du-caminhos, no pior caso.

OBS8: O grafo de fluxo de controle da Fig. B tem, no pior caso, $((11/2) * t + 9) * 2^t - 10 * t - 9$ du-caminhos.

Lema A 1 *Um grafo de fluxo de controle, correspondente a um programa com t comandos de decisão com no máximo dois sucessores execucionais, tem no maximo 2^t caminhos completos livres de laços.*

Proof: A prova é imediata ([MAL88b]).

Lema A 2 *Um grafo de fluxo de controle, correspondente a uma sequência de comandos IF-THEN-ELSE, maximiza o número de caminhos completos livre de laços, dado por $nc_t = 2^t$.*

Proof: A prova é immediata ([MAL88b]).

Lema A 3 *Um grafo de fluxo de controle, correspondente a uma sequência de comandos IF-THEN-ELSE, maximiza o número de du-caminhos para grafos de fluxo de controle sem laços, dado por $2^{t+4} - 10 * t - 16$.*

Prova: Considere um grafo de fluxo de controle G_{t-1} sem laços, com $t-1$ comandos de decisão, ilustrado na Fig. B. Suponha que G_{t-1} maximize o número de du-caminhos, no pior caso, dado por ndu_{t-1}. $(M_i)_t$ representa o número de du-caminhos com início no nó i. Seja $(M_1)_t$ $(M_2)_t$ tal que $(M_1)_t \geq (M_2)_t \geq (M_i)_t, \forall i, 2 < i \leq f(t)$, onde $f(t)$ fornece o número de nós do grafo. $(n_i)_t$ representa o número de du-caminhos terminando no nó i. Seja $(N_1)_t$ tal que $(N_1)_t \geq (n_i)_t, \forall i, 1 < i \leq f(t)$.

As únicas alternativas para introduzir uma estrutura, correspondente a um comando de decisão, no grafo G_{t-1} estão ilustradas na Fig. B. Obviamente, as três últimas podem ser eliminadas. É fácil verificar, também, que a primeira alternativa é uma combinação da segunda e terceira alternativas. Da Fig. B, pode-se obter:

$$ndu_t \leq 6 + 4 * (M_1)_{t-1} + ndu_{t-1}.$$

semelhantemente,

$$ndu_{t-1} \leq 6 + 4 * (M_1)_{t-2} + ndu_{t-2}$$

$$\vdots$$

$$ndu_2 \leq 6 + 4 * (M_1)_1 + ndu_1$$

.

Obtém-se também que,

$(M_1)_1 \leq 4$ e $ndu_1 \leq 6$

e, no pior caso

$(M_1)_1 = 4$ e $ndu_1 = 6$.

Adicionalmente, têm-se que

$$(M_1)_t = 4 + 2 * (M_1)_{t-1}$$

Combinando-se as expressões acima, têm-se:

$$ndu_t \leq 6 * (t-1) + 4 * ((M_1)_{t-1} + (M_1)_{t-2} + \ldots + (M_1)_1) + ndu_1(I)$$

e

$$(M_1)_t = 2^{t+2} - 4(II)$$

(I) and $(II) \Rightarrow ndu_t \leq 2^{t+4} - 10 * t - 16$.

Da mesma forma, pode-se obter:

$$ndu_t \leq 6 * (t-1) + 4 * ((N_1)_{t-1} + (N_1)_{t-2} + \ldots + (N_1)_1) + ndu_1(III)$$

e

$$(N_1)_t = 2^{t+2} - 4(IV)$$

(III) and $(IV) \Rightarrow ndu_t \leq 2^{t+4} - 10 * t - 16$.

Devido à OBS6, pode-se concluir que o grafo de fluxo de controle correspondente a uma sequência de comandos IF-THEN-ELSE maximiza o número de du-caminhos, para grafos de fluxo de controle livres de laços. •

Teorema A 1 *O grafo de fluxo de controle da Fig. B maximiza o nùmero de du-caminhos, consequentemente, o número de potencial-du-caminhos (Lema2), dado por* $((11/2) * t + 9) * 2^t - 10 * t - 9$.

Prova[1] *:*

Devido ao Lema A3 e às observações OBS1, OBS6 and OBS7, pode-se inferir que a partir de um grafo de fluxo de controle G_{t-1}, correspondente a uma seqüência de $(t-1)$ comandos IF-THEN-ELSE, a única opção para adicionar uma estrutura de decisão, com o objetivo de maximizar o número de du-caminhos, é adicionar uma estrutura de iteração; as alternativas para adicionar uma estrutura de iteração no grafo G_{t-1} são ilustradas na Fig. B. Analisando essas alternativas, pode-se concluir que a primeira alternativa maximiza o número de du-caminhos; neste caso, o número de du-caminhos adicionais é:

$$1 + 4 * (2^t - 1) + 3 * 2^{t-1} +$$

$$+3 * 2^{t-1} + 3 * 2 * 2^{t-2} + 3 * 2^{t-2} + 3 * 2 * 2^{t-3} + \ldots + 3 * 2^1 + 3 * 2 * 2^0 + 3 * 2^0 +$$

$$+2^{t-2} * 4 * (2^1 - 1) + 2^{t-3} * 4 * (2^2 - 1) + \ldots + 2^1 * 4 * (2^{t-2} - 1) + 2^0 * 4 * (2^{t-1} - 1) +$$

$$+2 * 2^{t-2} * 4 * (2^0 - 1) + 2 * 2^{t-3} * 4 * (2^1 - 1) + \ldots + 2 * 2^1 * 4 * (2^{t-3} - 1) + 2 * 2^0 * 4 * (2^{t-2} - 1) +$$

$$+2 * 2^{t-2} * 2^0 * 3 + 2 * 2^{t-3} * 2^1 * 3 + \ldots + 2 * 2^1 * 2^{t-3} * 3 + 2 * 2^0 * 2^{t-2} * 3 +$$

$$+1 + 4 * (2^{t-1} - 1) + 2 * 2^{t-1}.$$

Para o grafo todo têm-se:

$2^{t+3} - 10 * t - 9 + 2^{t+1} + 3 * 2^{t-1} - 3 + 3 * 2^{t+1} - 6 + (t-2) * 2^{t+1} + 4 + (t-3) * 2^{t+1} + 8 + 3 * (t-1) * 2^{t-1} + 6 * 2^{t-1} - 3 = ((11/2) * t + 9) * 2^t - 10 * t - 9.$

Observe que esta estrutura corresponde ao grafo de fluxo de controle da Fig. B.•

OBS: Fig. B ilustra o grafo de fluxo de controle que maximiza o numero de du-caminhos para programas que nunca terminam ("never-ending programs").

[1]Os autores agradecem a Christiano Lira Filho and Amir Said pelas sugestões e direções dadas.

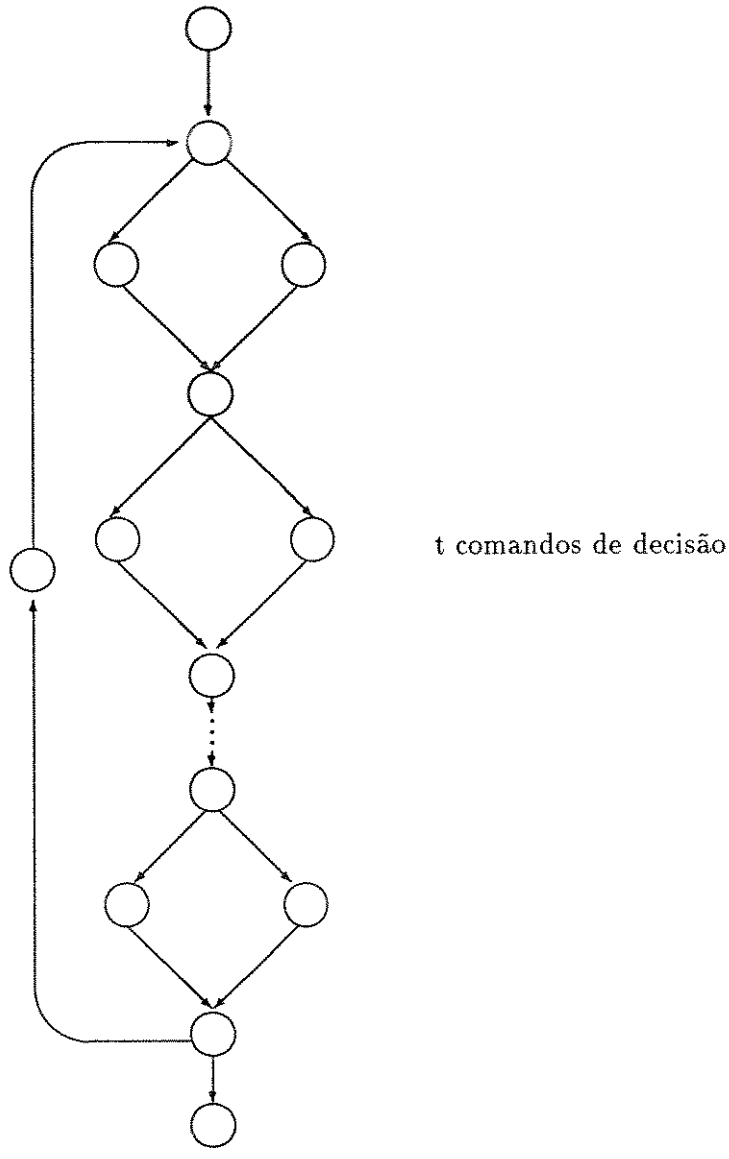


Figura B.1: Estrutura de Controle que Maximiza o Número de Potenciais-du-caminhos

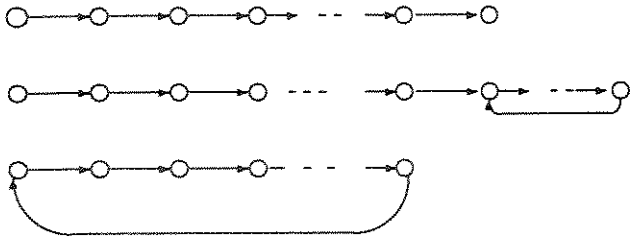


Figura B.2: Esquema de Du-caminhos

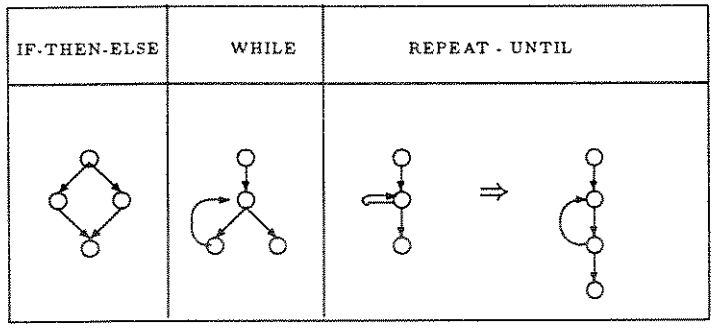


Figura B.3: Representação das Estruturas de Controle Básicas

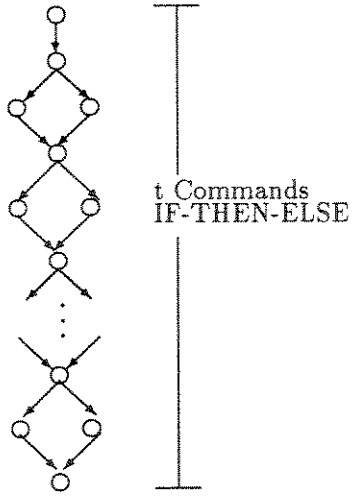


Figura B.4: Seqüência de t Estruturas de Controle IF-THEN-ELSE

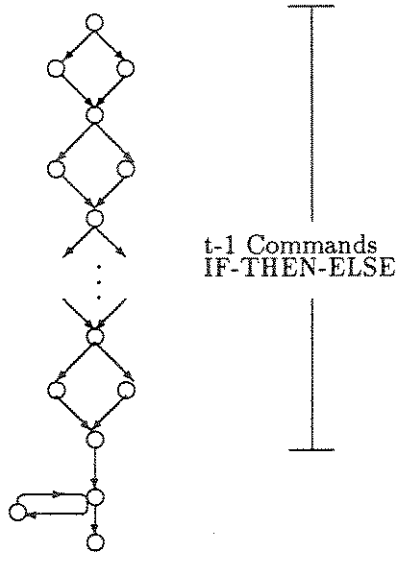
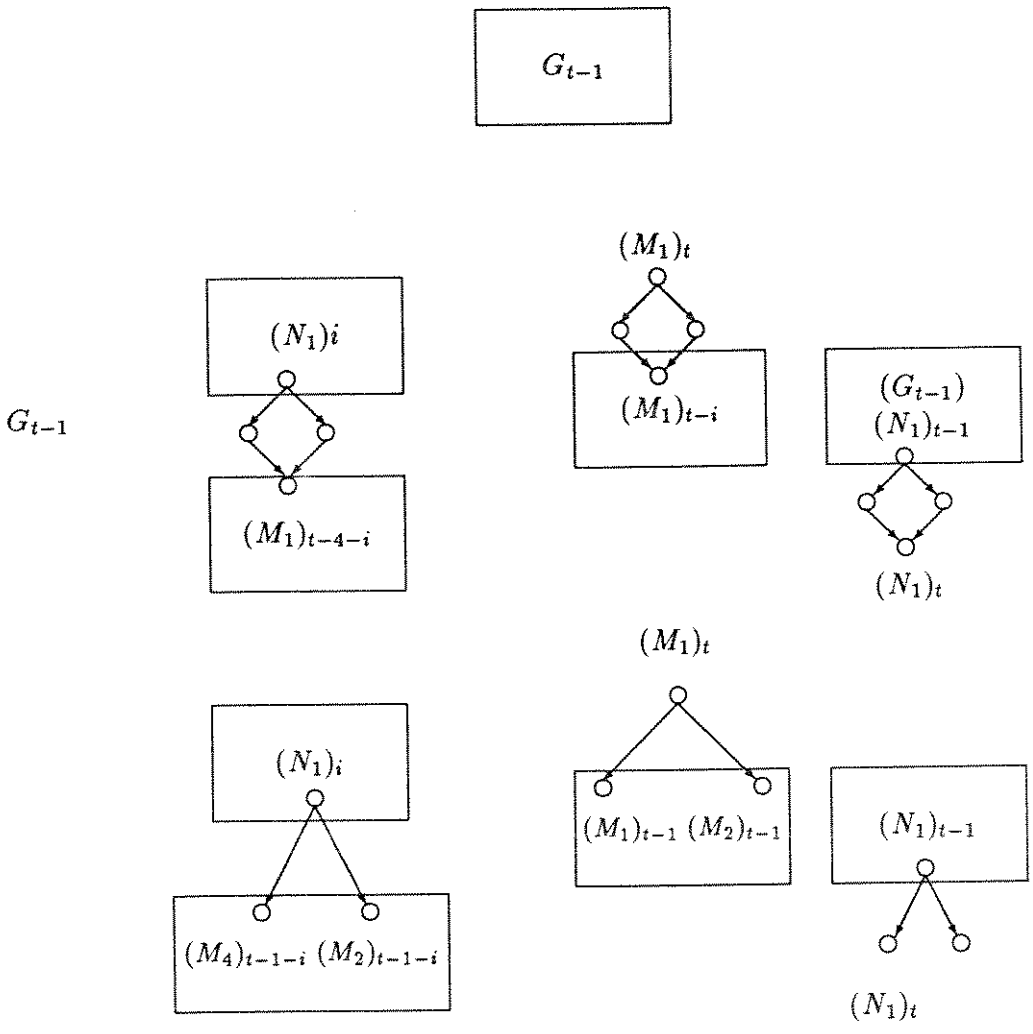
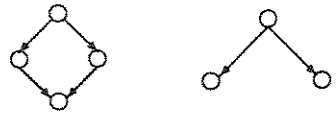


Figura B.5: Seqüência de t-1 Estruturas de Controle IF-THEN-ELSE Seguida de uma Estrutura de Iteração



for t=1



$(M_1)_1 \leq 4$
$(N_1)_1 \leq 4$
$ndu_1 \leq 6$

pior caso:

$(M_1)_1 = 4$
$(N_1)_1 = 4$
$ndu_1 = 6$

Figura B.6: Alternativas para Adicionar uma Estrutura de Decisão a um Grafo sem Laços

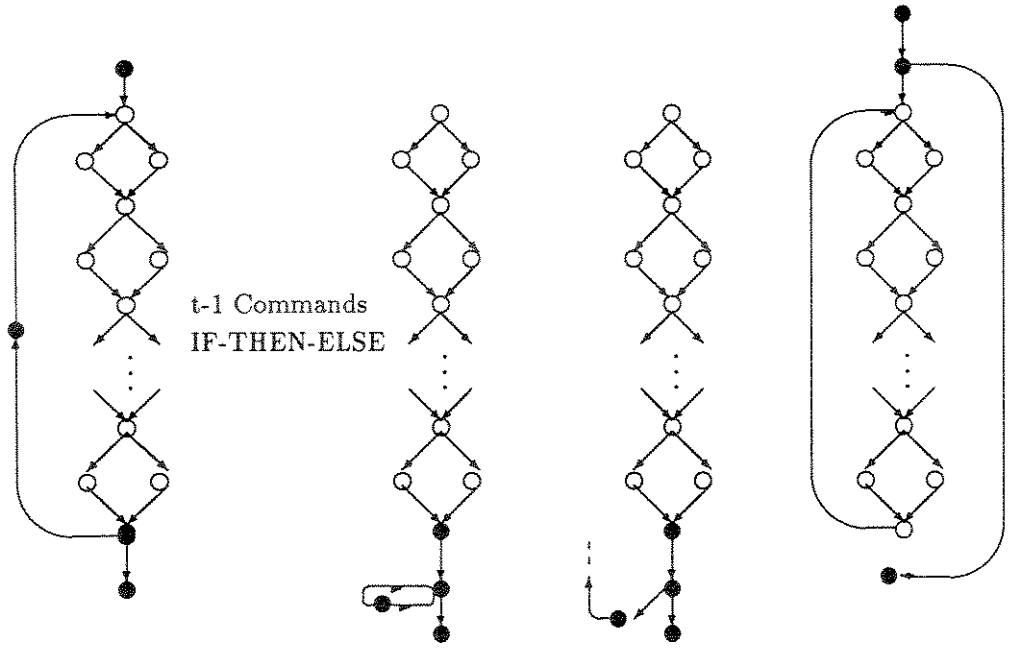


Figura B.7: Alternativas para Adicionar uma Estrutura de Iteração em umaa Seqüência de t-1 Estruturas IF-THEN-ELSE

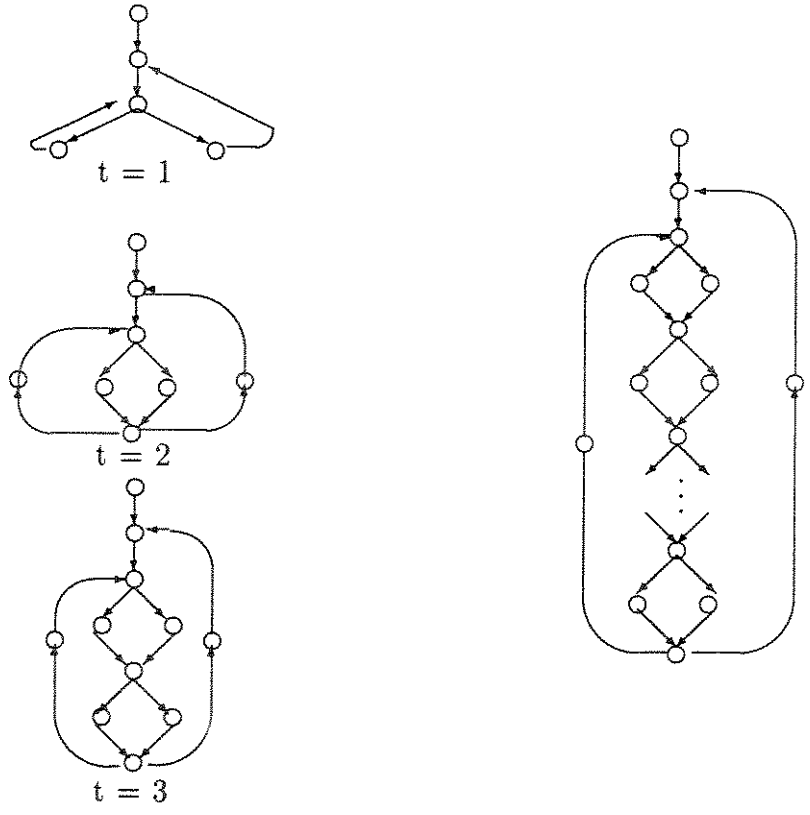


Figura B.8: Grafo de Fluxo de Controle que Maximiza o Número de Potenciais-du-caminhos para Programas que Nunca Terminam ("never-ending programs").

Apêndice C

Um Exemplo Completo

Neste apêndice é apresentado um exemplo completo, extraído do "benchmark", utilizado para ilustrar as informações estáticas e dinâmicas geradas pela ferramenta POKE-TOOL.

O programa apresentado aqui é chamado **entab** (pág. 32, [KER81]) e sua função é copiar a entrada, fornecida via teclado, para a saída, substituindo cadeias de espaços em branco por caracteres de tabulação de tal maneira que visualmente a saída é igual à entrada, porém com menor número de caracteres. Este programa encontra-se no arquivo ENTAB.C (ver Seção C.1); a partir de agora, utilizaremos o nome de seu arquivo para referenciarmos este programa.

Para distinguir entre telas e mensagens do sistema, entradas do usuário e comentários, foi adotada a seguinte convenção: o que for relativo à ferramenta será descrito em "font" de máquina de escrever, as entradas do usuário são impressas em negrito e os comentários a respeito do funcionamento da ferramenta em itálico.

A maioria das respostas da ferramenta aparece na forma de arquivos que são apresentados ao usuário através do utilitário *more*; estes arquivos estão organizados nas Seções C.1 e C.2 seguintes, que contêm, respectivamente, arquivos referentes às informações estáticas e dinâmicas geradas pela POKE-TOOL.

A seguir dá-se uma idéia de utilização da POKE-TOOL, apresentando-se suas principais telas; uma descrição detalhada de sua operação e utilização é fornecida em [CHA91?, CHA91?].

A tela INICIAÇÃO é a primeira tela apresentada ao usuário. Caso 'N' ou 'n' seja teclado, a POKE-TOOL solicita ao usuário o diretório onde se encontra os dados da sessão de trabalho a ser recuperada. O símbolo que encontra-se entre colchetes é considerado "default", ou seja, qualquer tecla, a menos daquelas que selecionam a outra opção, provocam a sua seleção. Neste exemplo, é iniciada uma sessão de teste desde o princípio.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

 Bem-vindo !!

                              INICIACAO

Voce deseja iniciar uma nova sessao de trabalho (S/N) [S] ?

==>s


Mensagens-------------------------------------------------------------

POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

 Bem-vindo !!

                              INICIACAO

Posso apagar os arquivos que estao nesse diretorio (S/N) [N] ?

==>s


Mensagens-------------------------------------------------------------

* * Apagando arquivos da ultima sessao de trabalho * *
```

Iniciada a sessão de trabalho e fornecido o nome da unidade a ser testada, a POKE-TOOL dá início à fase estática da sessão de trabalho que consiste na análise do código fonte através dos módulos **li**, **chanomat** e **pokernel**, *conforme ilustrado a seguir.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Vers.0.2

 Bem-vindo !!

                              INICIACAO

Entre com o nome do arquivo que contem a unidade a ser testada (digite
"fim" para terminar a sessao):

==>entab.c


Mensagens-------------------------------------------------------------

* * Determinando o Grafo de Fluxo de Controle * *

* * Ferramenta Geradora de GFC's foi bem sucedida * *

* * Calculando os arcos primitivos ... * *

* * Carregando Tabela de Transicao lexica ... * *

* * Fazendo a analise sintatica do codigo fonte ... * *

* * Gerando descritores ... * *

* * O Nucleo da POKE-TOOL foi bem sucedido * *
```

Neste ponto os arquivos ENTAB.LI, ENTAB.NLI, ENTAB.GFC, ARCPRIM.TES, GRAFODEF.TES, TESTEPROG.C, PUASSOC.TES, PDUPATHS.TES, DES_PU.TES, DES_PUDU e DES_PDU já foram criados pela POKE-TOOL. Terminado o processo de preparação da unidade em teste a tela MENU PRINCIPAL é apresentada ao usuário, indicando que a fase de teste propriamente dita pode ser iniciada, ou seja, a fase dinâmica da sessão de trabalho pode ter início. Esta tela apresenta tanto os resultados da fase estática como os da fase dinâmica.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                              MENU PRINCIPAL

a. Visualizacao Arquivos
b. Gera Programa Executavel
c. Executa Caso de Teste
d. Avaliacao Caso de Teste
e. Termina Sessao

Entre com a opcao desejada:

==>a


Mensagens-------------------------------------------------------------
```

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                          VISUALIZACAO DE ARQUIVOS

a. Arcos Primitivos                    i. Caminhos percorridos pelos Casos
b. Grafo Def                              de Teste
c. Arquivo Modificado                  j. Associacoes executadas para o
d. Potenciais Du-caminhos Requeridos      criterio todos-potenciais-usos
e. Associacoes Requeridas              k. Associacoes executadas para o
f. Entrada dos Casos de teste             criterio todos-potenciais-usos/du
g. Entrada do teclado                  l. Potenciais-du-caminhos executados
h. Saida dos Casos de Teste            m. Retorna para o Menu Principal

Entre com a opcao desejada:

==>


Mensagens-------------------------------------------------------------
```

As opções **a**, **b**, **c**, **d** *e* **e** *apresentam os resultados da análise estática; estão associadas, respectivamente, aos arquivos, gerados pela POKE-TOOL, ARCPRIM.TES, GRAFODEF.TES, TESTEPROG.TES, PDUPATHS.TES e PUASSOC.TES incluídos na Seção C.1. As demais opções estão associadas à fase dinâmica da ferramenta e são*

arquivos gerados pela execução e avaliação dos casos de teste. No atual ponto da sessão de trabalho nenhum desse arquivos foi gerado ainda, pois nenhum caso de teste foi executado, nem avaliado.

Para executar e avaliar casos de teste é necessário retornar à tela MENU PRINCIPAL; porém, antes de executar um caso de teste é necessário gerar o programa executável que contém a unidade em teste instrumentada. selecionado-se a opção **b** *do MENU PRINCIPAL, obtendo-se a tela GERA PROGRAMA EXECUTÁVEL, ilustrada a seguir.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                    GERA PROGRAMA EXECUTAVEL


OBSERVACAO: Para gerar o programa executavel para teste, voce recebe
            o "prompt" do sistema operacional. Voce devera' compilar
            e "linkar" o arquivo TESTEPROG.C junto com os seus
            outros arquivos no lugar do modulo a ser testado. O novo
            programa executavel devera' ter o nome TESTEPROG.

Digite qualquer tecla para entrar no "shell" do sistema operacional...

Para retornar 'a POKE-TOOL digite "exit"...
```

Ao retornar à POKE-TOOL retorna-se à tela MENU PRINCIPAL.

Para executar um caso de teste basta selecionar a opção **c** *e a tela EXECUTA CASO DE TESTE será apresentada.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                        EXECUTA CASO DE TESTE

Seu programa necessita de parametros de entrada (S/N) [N]?

==>n

Seu programa tem entrada pelo teclado (S/N) [N]?

==>s


Mensagens-------------------------------------------------------------

* * A saida esta' sendo direcionada para o arquivo output.tes * *
```

Na primeira vez que se executa um caso de teste a tela EXECUTA CASO DE TESTE pergunta se o programa utiliza parâmetros de entrada na linha de comandos e entrada via teclado; as respostas a essas perguntas são associadas à unidade em teste e utilizadas na execução de outros casos de teste na sessão corrente ou em outras sessões de trabalho.

```
col 1   2   34  rest

col 1 2 34 rest
Tecle qualquer tecla para retornar 'a POKE-TOOL ...
```

A unidade ENTAB.C não aceita parâmetros de entrada através da linha de comandos mas, se aceitasse, estes parâmetros de entrada seriam salvos no arquivo INPUT1.TES. A entrada via teclado, a saída na tela do caso de teste e o caminho percorrido pelos casos de teste são salvos, respectivamente, nos arquivos TEC1.TES, OUTPUT1.TES e PATH1.TES. O número indica que são arquivos associados ao primeiro caso de teste. As demais entradas via teclado, saídas na tela e caminhos percorridos para os outros casos de teste executados para o programa ENTAB.C são apresentados na Seção C.2. Depois de executado um caso de teste a tela MENU PRINCIPAL é reapresentada e o usuário pode decidir entre executar mais um caso de teste ou avaliar os já executados com relação a algum critério PU. Selecionando-se a opção d *do menu principal, obtém-se a tela AVALIAÇÃO, ilustrada a seguir.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                              AVALIACAO

a. Criterio Todos-Potenciais-Usos

b. Criterio Todos-Potenciais-Usos/du

c. Criterio Todos-Potenciais-Du-Caminhos

d. Retorna para Menu Principal

Entre com a opcao desejada:

==>a

Mensagens-------------------------------------------------------------

* * Realizando a avaliacao do caso de teste * *

* * Avaliacao do caso de teste foi bem sucedida * *

* * Terminada a avaliacao de um caso de teste * *
```

O usuário tem três critérios PU implementados na POKE-TOOL que ele pode selecionar; ao selecionar um dos critérios, por exemplo, todos-potenciais-usos, faz com que a POKE-TOOL dê início ao processo de avaliação. Durante a avaliação a ferramenta envia mensagens que indicam o andamento do processo. Terminada a avaliação, a POKE-TOOL apresenta o arquivo PUOUTPUT.TES, para o critério em questão, que contém as associações não executadas pelo conjunto de casos de teste avaliado. Depois de apresentado o arquivo, a tela AVALIAÇÃO é reapresentada. Se o usuário selecionar os critérios todos-potenciais-usos e todos-potenciais-du-caminhos para avaliação do conjunto de casos de teste, os arquivos PUDUOUTPUT.TES e PDUOUTPUT.TES são, respectivamente, apresentados ao final da avaliação. Na Seção C.2 os arquivos PUOUTPUT.TES, PUDUOUTPUT.TES e PDUOUTPUT.TES ilustram os resultados obtidos após a avaliação de oito casos de teste executados para o exemplo ENTAB.C.

Neste ponto já é possível observar os resultados obtidos da fase dinâmica da sessão de trabalho. As opções **f**, **g**, **h** *e* **i**, *da tela VISUALIZAÇÃO DE ARQUIVOS, apresentam, respectivamente, os parâmetros de entrada, entradas via teclado, as saídas na tela e os caminhos percorridos pelos casos de teste. Porém, essas opções estão relacionadas com vários arquivos porque existe um arquivo para cada caso de teste. Quando o usuário seleciona, por exemplo, a entrada do teclado, a POKE-TOOL pergunta a qual*

caso de teste o usuário está se referindo; se ele selecionar o sétimo caso de teste, a POKE-TOOL irá apresentar o arquivo TEC7.TES. Fato análogo ocorre com as outras opções acima e os conjuntos de arquivos associados.

As opções **j**, **k**, **l** *apresentam as associações e caminhos efetivamente exercitados pelos casos de teste avaliados. Os arquivos associados com essas opções são, respectivamente, EXEC_PU.TES, EXEC_PUDU.TES e EXEC_PDU.TES. Essas opções apresentam os resultados da última avaliação efetuada; pode ocorrer de terem sido executados casos de teste adicionais que não foram avaliados; portanto, para apresentar o resultado atualizado deve-se primeiramente avaliar os casos de teste. A Seção C.2 contém os arquivos acima obtidos para o exemplo ENTAB.C depois da avaliação de oito casos de teste.*

Selecionando-se a opção **e** *do menu principal é iniciado o processo de término da sessão. A POKE-TOOL pergunta ao usuário se ele deseja que os arquivos gerados pela ferramenta sejam salvos em um sub-diretório com o nome da unidade; em caso afirmativo, a POKE-TOOL cria, se necessário, este sub-diretório e salva os arquivos nele; em caso contrário, a POKE-TOOL termina a execução sem salvar as informações geradas. Essas informações são sempre salvas no diretório corrente, independentemente da opção do usuário.*

C.1 Informações Estáticas

Nesta seção são organizadas as principais informações estáticas geradas pela POKE-TOOL, para o programa ENTAB.C; estas informações estão contidas em arquivos gerados pela POKE-TOOL, durante a fase de análise estática e são reproduzidos a seguir, sem qualquer alteração.

Arquivo Entab.c

```
#include "cte.c"

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);
}
```

Arquivo Entab.nli

```
$DCL                    1      1    181      1
{                       1    182      1     13
$DCL                    1    189     17     15
$DCL                    1    209     22     16
$S1                     1    234     18     17
$S2                     1    255      8     18
$REPEAT                 2    268      2     20
{                       2    273      1     21
$S3                     2    279     13     22
$WHILE                  3    297      5     23
$C(01)1                 3    303     26     23
{                       4    334      1     24
$S4                     4    342      9     25
$IF                     4    358      2     26
$C(01)2                 4    361     25     26
{                       5    395      1     27
$S5                     5    407     13     28
$S6                     5    431     13     29
}                       5    453      1     30
}                       6    459      1     31
$WHILE                  7    465      5     32
$C(01)3                 7    471     12     32
{                       8    488      1     33
$S7                     8    495     15     34
$S8                     8    516      6     35
}                       8    527      1     36
$IF                     9    533      2     37
$C(01)4                 9    536     12     37
{                      10    553      1     38
$S9                    10    561     11     39
$IF                    10    579      2     40
$C(01)5                10    582     12     40
{                      11      0      0      0
$S10                   11    603      8     41
}                      11      0      0      0
$ELSE                  12    618      4     42
{                      12      0      0      0
```

```
$S11                    12    631      6     43
}                       12      0      0      0
}                       13    642      1     44
}                       14    646      1     45
$UNTIL                  14    647      5     45
$NC(01)6                14    653     12     45
}                       15    668      0     46
```

Arquivo Entab.gfc

```
15

1
   2 0
2
   3 0
3
   4 7 0
4
   5 6 0
5
   6 0
6
   3 0
7
   8 9 0
8
   7 0
9
   10 14 0
10
   11 12 0
11
   13 0
12
   13 0
13
   14 0
14
   2 15 0
15
   0
```

Arquivo arcprim.tes

```
ARCOS PRIMITIVOS DO MODULO entab.c

arco ( 4, 5) e' primitivo
arco ( 4, 6) e' primitivo
arco ( 7, 8) e' primitivo
arco ( 9,14) e' primitivo
arco (10,11) e' primitivo
```

```
arco (10,12) e' primitivo
arco (14, 2) e' primitivo
arco (14,15) e' primitivo
```

Arquivo Testeprog.c

```
#define ponta_de_prova(num) if(printed_nodes % 10) {++printed_nodes;
fprintf(path," %d ",num);}\ else {++printed_nodes; fprintf(path,
"%d\n",num);}

#include <stdio.h>


#include "cte.c"



void entab()

/*  Substitui strings de brancos por tabs. Produz visualmente a
    mesma saida, mas com menos caracteres                          */



/*  1 */        {
                   FILE * path = fopen("path.tes","w");
                   static int printed_nodes = 0;
/*  1 */           int c,col,newcol;
/*  1 */           int tabstops[MAXLINE];
                   ponta_de_prova(1);
/*  1 */           settabs(tabstops);
/*  1 */           col = 0;
/*  2 */           do
/*  2 */           {
                      ponta_de_prova(2);
/*  2 */              newcol = col;
/*  3 */              while((c = getchar()) == BLANK)
/*  4 */              {
                         ponta_de_prova(3);
                         ponta_de_prova(4);
/*  4 */                 newcol++;
/*  4 */                 if(tabpos(newcol,tabstops))
/*  5 */                 {
                            ponta_de_prova(5);
/*  5 */                    putchar(TAB);
/*  5 */                    col = newcol;
/*  5 */                 }
                         ponta_de_prova(6);
```

```
/*  6 */                }
                        ponta_de_prova(3);
/*  7 */                while(col<newcol)
/*  8 */                {
                           ponta_de_prova(7);
                           ponta_de_prova(8);
/*  8 */                   putchar(BLANK);
/*  8 */                   col++;
/*  8 */                }
                        ponta_de_prova(7);
                        ponta_de_prova(9);
/*  9 */                if(c!=ENDFILE)
/* 10 */                {
                           ponta_de_prova(10);
/* 10 */                   putchar(c);
/* 10 */                   if(c==NEWLINE)
/* 11 */                   {
                              ponta_de_prova(11);
/* 11 */                      col = 0;
/* 11 */                   }
/* 12 */                   else
/* 12 */                   {
                              ponta_de_prova(12);
/* 12 */                      col++;
/* 12 */                   }
                           ponta_de_prova(13);
/* 13 */                }
                        ponta_de_prova(14);
/* 14 */             }
/* 14 */             while(c!=ENDFILE);
                     ponta_de_prova(15);
                     fclose(path);
/* 15 */          }
```

Arquivo grafodef.tes

```
VARIAVEIS DEFINIDAS nos NOS do modulo entab.c

Variaveis Definidas  = Vars Defs
Variaveis possivelmente definidas por Referencia = Vars Refs

NO'  1
Vars defs:  col
Vars refs:  tabstops
NO'  2
Vars defs:  newcol
Vars refs:
NO'  3
Vars defs:  c
Vars refs:
NO'  4
Vars defs:  newcol
Vars refs:  tabstops
NO'  5
Vars defs:  col
Vars refs:
NO'  6
Vars defs:
Vars refs:
NO'  7
Vars defs:
Vars refs:
NO'  8
Vars defs:  col
Vars refs:
NO'  9
Vars defs:
Vars refs:
NO' 10
Vars defs:
Vars refs:
NO' 11
Vars defs:  col
Vars refs:
NO' 12
Vars defs:  col
Vars refs:
NO' 13
Vars defs:
```

```
Vars refs:
NO' 14
Vars defs:
Vars refs:
NO' 15
Vars defs:
Vars refs:
```

Arquivo puassoc.tes

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


Associacoes requeridas pelo Grafo( 1)

1) <1,(9,14),{ col, tabstops }>
2) <1,(10,12),{ col, tabstops }>
3) <1,(14,15),{ col, tabstops }>
4) <1,(14,15),{ tabstops }>
5) <1,(14,2),{ col, tabstops }>
6) <1,(14,2),{ tabstops }>
7) <1,(10,11),{ col, tabstops }>
8) <1,(8,7),{ tabstops }>
9) <1,(7,8),{ col, tabstops }>
10) <1,(4,6),{ col, tabstops }>
11) <1,(6,3),{ col, tabstops }>
12) <1,(6,3),{ tabstops }>
13) <1,(4,5),{ col, tabstops }>


Associacoes requeridas pelo Grafo( 2)

14) <2,(9,14),{ newcol }>
15) <2,(10,12),{ newcol }>
16) <2,(14,15),{ newcol }>
17) <2,(14,2),{ newcol }>
18) <2,(10,11),{ newcol }>
19) <2,(8,7),{ newcol }>
20) <2,(7,8),{ newcol }>
21) <2,(3,4),{ newcol }>


Associacoes requeridas pelo Grafo( 3)

22) <3,(9,14),{ c }>
23) <3,(10,12),{ c }>
24) <3,(14,15),{ c }>
25) <3,(2,3),{ c }>
26) <3,(14,2),{ c }>
27) <3,(10,11),{ c }>
28) <3,(8,7),{ c }>
29) <3,(7,8),{ c }>
30) <3,(4,6),{ c }>
```

```
31) <3,(6,3),{ c }>
32) <3,(4,5),{ c }>


Associacoes requeridas pelo Grafo( 4)

33) <4,(4,6),{ newcol, tabstops }>
34) <4,(9,14),{ newcol, tabstops }>
35) <4,(10,12),{ newcol, tabstops }>
36) <4,(14,15),{ newcol, tabstops }>
37) <4,(2,3),{ tabstops }>
38) <4,(14,2),{ newcol, tabstops }>
39) <4,(10,11),{ newcol, tabstops }>
40) <4,(8,7),{ newcol, tabstops }>
41) <4,(7,8),{ newcol, tabstops }>
42) <4,(3,4),{ newcol, tabstops }>
43) <4,(4,5),{ newcol, tabstops }>


Associacoes requeridas pelo Grafo( 5)

44) <5,(14,15),{ col }>
45) <5,(2,3),{ col }>
46) <5,(14,2),{ col }>
47) <5,(9,14),{ col }>
48) <5,(10,12),{ col }>
49) <5,(10,11),{ col }>
50) <5,(7,8),{ col }>
51) <5,(4,6),{ col }>
52) <5,(4,5),{ col }>


Associacoes requeridas pelo Grafo( 8)

53) <8,(14,15),{ col }>
54) <8,(3,7),{ col }>
55) <8,(6,3),{ col }>
56) <8,(4,6),{ col }>
57) <8,(4,5),{ col }>
58) <8,(14,2),{ col }>
59) <8,(9,14),{ col }>
60) <8,(10,12),{ col }>
61) <8,(10,11),{ col }>
62) <8,(7,8),{ col }>
```

```
Associacoes requeridas pelo Grafo(11)

63) <11,(14,15),{ col }>
64) <11,(9,14),{ col }>
65) <11,(10,12),{ col }>
66) <11,(10,11),{ col }>
67) <11,(7,8),{ col }>
68) <11,(6,3),{ col }>
69) <11,(4,6),{ col }>
70) <11,(4,5),{ col }>
71) <11,(14,2),{ col }>


Associacoes requeridas pelo Grafo(12)

72) <12,(14,15),{ col }>
73) <12,(9,14),{ col }>
74) <12,(10,12),{ col }>
75) <12,(10,11),{ col }>
76) <12,(7,8),{ col }>
77) <12,(6,3),{ col }>
78) <12,(4,6),{ col }>
79) <12,(4,5),{ col }>
80) <12,(14,2),{ col }>
```

Arquivo pdupaths.tes

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS


Caminhos requeridos pelo Grafo( 1)

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


Caminhos requeridos pelo Grafo( 2)

10) 2 3 7 9 14 15
11) 2 3 7 9 14 2
12) 2 3 7 9 10 12 13 14 15
13) 2 3 7 9 10 12 13 14 2
14) 2 3 7 9 10 11 13 14 15
15) 2 3 7 9 10 11 13 14 2
16) 2 3 7 8 7
17) 2 3 4


Caminhos requeridos pelo Grafo( 3)

18) 3 7 9 14 15
19) 3 7 9 14 2 3
20) 3 7 9 10 12 13 14 15
21) 3 7 9 10 12 13 14 2 3
22) 3 7 9 10 11 13 14 15
23) 3 7 9 10 11 13 14 2 3
24) 3 7 8 7
25) 3 4 6 3
26) 3 4 5 6 3


Caminhos requeridos pelo Grafo( 4)
```

```
27) 4 6 3 7 9 14 15
28) 4 6 3 7 9 14 2 3
29) 4 6 3 7 9 10 12 13 14 15
30) 4 6 3 7 9 10 12 13 14 2 3
31) 4 6 3 7 9 10 11 13 14 15
32) 4 6 3 7 9 10 11 13 14 2 3
33) 4 6 3 7 8 7
34) 4 6 3 4
35) 4 5 6 3 7 9 14 15
36) 4 5 6 3 7 9 14 2 3
37) 4 5 6 3 7 9 10 12 13 14 15
38) 4 5 6 3 7 9 10 12 13 14 2 3
39) 4 5 6 3 7 9 10 11 13 14 15
40) 4 5 6 3 7 9 10 11 13 14 2 3
41) 4 5 6 3 7 8 7
42) 4 5 6 3 4


Caminhos requeridos pelo Grafo( 5)

43) 5 6 3 7 9 14 15
44) 5 6 3 7 9 14 2 3
45) 5 6 3 7 9 10 12
46) 5 6 3 7 9 10 11
47) 5 6 3 7 8
48) 5 6 3 4 6
49) 5 6 3 4 5


Caminhos requeridos pelo Grafo( 8)

50) 8 7 9 14 15
51) 8 7 9 14 2 3 7
52) 8 7 9 14 2 3 4 6 3
53) 8 7 9 14 2 3 4 5
54) 8 7 9 10 12
55) 8 7 9 10 11
56) 8 7 8


Caminhos requeridos pelo Grafo(11)

57) 11 13 14 15
58) 11 13 14 2 3 7 9 14
```

59) 11 13 14 2 3 7 9 10 12
60) 11 13 14 2 3 7 9 10 11
61) 11 13 14 2 3 7 8
62) 11 13 14 2 3 4 6 3
63) 11 13 14 2 3 4 5

Caminhos requeridos pelo Grafo(12)

64) 12 13 14 15
65) 12 13 14 2 3 7 9 14
66) 12 13 14 2 3 7 9 10 12
67) 12 13 14 2 3 7 9 10 11
68) 12 13 14 2 3 7 8
69) 12 13 14 2 3 4 6 3
70) 12 13 14 2 3 4 5

Arquivo des_pu.tes

```
DESCRITORES PARA O CRITERIO TODOS POT-USOS

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


Descritores para o Grafo( 1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 2 3 7 15

1) N* 1 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 2 3 4 6 7 9 10 13 14 15

2) N* 1 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 6 7 9 10 13 14 15

3) N* 1 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 6 7 9 10 13 14 15

4) N* 1 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

5) N* 1 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 3 4 6 7 9 10 13 14 15

6) N* 1 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

7) N* 1 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 6 7 9 10 13 14 15

8) N* 1 Nnv* 8 [ Nnv* 8 ]* 7
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

9) N* 1 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 2 3 4 6 7 9 10 13 14 15

10) N* 1 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 6 7 9 10 13 14 15

11) N* 1 Nnv* 6 [ Nnv* 6 ]* 3
Nnv = 2 3 4 6 7 9 10 13 14 15
```

```
12) N* 1 Nnv* 6 [ Nnv* 6 ]* 3
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

13) N* 1 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 6 7 9 10 13 14 15



Descritores para o Grafo( 2)

Ni = 2 3 4 7 8 9 10 11 12 13 14 15
Nt = 2 4 7 15

14) N* 2 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 3 7 8 9 10 11 12 13 14 15

15) N* 2 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 3 7 8 9 10 11 12 13 14 15

16) N* 2 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 3 7 8 9 10 11 12 13 14 15

17) N* 2 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 3 7 8 9 10 11 12 13 14 15

18) N* 2 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 3 7 8 9 10 11 12 13 14 15

19) N* 2 Nnv* 8 [ Nnv* 8 ]* 7
Nnv = 3 7 8 9 10 11 12 13 14 15

20) N* 2 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 3 7 8 9 10 11 12 13 14 15

21) N* 2 Nnv* 3 [ Nnv* 3 ]* 4
Nnv = 3 7 8 9 10 11 12 13 14 15



Descritores para o Grafo( 3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 7 15

22) N* 3 Nnv* 9 [ Nnv* 9 ]* 14
```

```
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

23) N* 3 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

24) N* 3 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

25) N* 3 Nnv* 2 [ Nnv* 2 ]* 3
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

26) N* 3 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

27) N* 3 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

28) N* 3 Nnv* 8 [ Nnv* 8 ]* 7
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

29) N* 3 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

30) N* 3 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

31) N* 3 Nnv* 6 [ Nnv* 6 ]* 3
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

32) N* 3 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15



Descritores para o Grafo( 4)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 4 7 15

33) N* 4 [ Nnv* 4 ]* 6
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

34) N* 4 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15
```

```
35) N* 4 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

36) N* 4 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

37) N* 4 Nnv* 2 [ Nnv* 2 ]* 3
Nnv = 2 3 5 6 7 8 9 10 11 12 13 14 15

38) N* 4 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

39) N* 4 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

40) N* 4 Nnv* 8 [ Nnv* 8 ]* 7
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

41) N* 4 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

42) N* 4 Nnv* 3 [ Nnv* 3 ]* 4
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

43) N* 4 [ Nnv* 4 ]* 5
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15


Descritores para o Grafo( 5)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 6 8 11 12 15

44) N* 5 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 6 7 9 10 14 15

45) N* 5 Nnv* 2 [ Nnv* 2 ]* 3
Nnv = 2 3 4 6 7 9 10 14 15

46) N* 5 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 3 4 6 7 9 10 14 15

47) N* 5 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 2 3 4 6 7 9 10 14 15
```

```
48) N* 5 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 6 7 9 10 14 15

49) N* 5 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 6 7 9 10 14 15

50) N* 5 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 2 3 4 6 7 9 10 14 15

51) N* 5 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 6 7 9 10 14 15

52) N* 5 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 6 7 9 10 14 15



Descritores para o Grafo( 8)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 7 8 11 12 15

53) N* 8 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 6 7 9 10 14 15

54) N* 8 Nnv* 3 [ Nnv* 3 ]* 7
Nnv = 2 3 4 6 7 9 10 14 15

55) N* 8 Nnv* 6 [ Nnv* 6 ]* 3
Nnv = 2 3 4 6 7 9 10 14 15

56) N* 8 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 6 7 9 10 14 15

57) N* 8 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 6 7 9 10 14 15

58) N* 8 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 3 4 6 7 9 10 14 15

59) N* 8 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 2 3 4 6 7 9 10 14 15

60) N* 8 Nnv* 10 [ Nnv* 10 ]* 12
```

Nnv = 2 3 4 6 7 9 10 14 15

61) N* 8 Nnv* 10 [Nnv* 10]* 11
Nnv = 2 3 4 6 7 9 10 14 15

62) N* 8 Nnv* 7 [Nnv* 7]* 8
Nnv = 2 3 4 6 7 9 10 14 15

Descritores para o Grafo(11)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

63) N* 11 Nnv* 14 [Nnv* 14]* 15
Nnv = 2 3 4 6 7 9 10 13 14 15

64) N* 11 Nnv* 9 [Nnv* 9]* 14
Nnv = 2 3 4 6 7 9 10 13 14 15

65) N* 11 Nnv* 10 [Nnv* 10]* 12
Nnv = 2 3 4 6 7 9 10 13 14 15

66) N* 11 Nnv* 10 [Nnv* 10]* 11
Nnv = 2 3 4 6 7 9 10 13 14 15

67) N* 11 Nnv* 7 [Nnv* 7]* 8
Nnv = 2 3 4 6 7 9 10 13 14 15

68) N* 11 Nnv* 6 [Nnv* 6]* 3
Nnv = 2 3 4 6 7 9 10 13 14 15

69) N* 11 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 3 4 6 7 9 10 13 14 15

70) N* 11 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 6 7 9 10 13 14 15

71) N* 11 Nnv* 14 [Nnv* 14]* 2
Nnv = 2 3 4 6 7 9 10 13 14 15

Descritores para o Grafo(12)

```
Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

72) N* 12 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 6 7 9 10 13 14 15

73) N* 12 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 2 3 4 6 7 9 10 13 14 15

74) N* 12 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 6 7 9 10 13 14 15

75) N* 12 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 6 7 9 10 13 14 15

76) N* 12 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 2 3 4 6 7 9 10 13 14 15

77) N* 12 Nnv* 6 [ Nnv* 6 ]* 3
Nnv = 2 3 4 6 7 9 10 13 14 15

78) N* 12 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 6 7 9 10 13 14 15

79) N* 12 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 6 7 9 10 13 14 15

80) N* 12 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 3 4 6 7 9 10 13 14 15


Numero Total de Descritores = 80
```

Arquivo des_pudu

```
DESCRITORES PARA O CRITERIO TODOS POT-USOS/DU

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


Descritores para o Grafo( 1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 2 3 7 15

1) N* 1 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 13 14 15

2) N* 1 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 13 14 15

3) N* 1 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 13 14 15

4) N* 1 Nnvlf* 14 15
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

5) N* 1 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 13 14 15

6) N* 1 Nnvlf* 14 2
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

7) N* 1 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 13 14 15

8) N* 1 Nnvlf* 8 7
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

9) N* 1 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 13 14 15

10) N* 1 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 13 14 15

11) N* 1 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 13 14 15
```

```
12) N* 1 Nnvlf* 6 3
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

13) N* 1 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 13 14 15


Descritores para o Grafo( 2)

Ni = 2 3 4 7 8 9 10 11 12 13 14 15
Nt = 2 4 7 15

14) N* 2 Nnvlf* 9 14
Nnvlf = 3 7 8 9 10 11 12 13 14 15

15) N* 2 Nnvlf* 10 12
Nnvlf = 3 7 8 9 10 11 12 13 14 15

16) N* 2 Nnvlf* 14 15
Nnvlf = 3 7 8 9 10 11 12 13 14 15

17) N* 2 Nnvlf* 14 2
Nnvlf = 3 7 8 9 10 11 12 13 14 15

18) N* 2 Nnvlf* 10 11
Nnvlf = 3 7 8 9 10 11 12 13 14 15

19) N* 2 Nnvlf* 8 7
Nnvlf = 3 7 8 9 10 11 12 13 14 15

20) N* 2 Nnvlf* 7 8
Nnvlf = 3 7 8 9 10 11 12 13 14 15

21) N* 2 Nnvlf* 3 4
Nnvlf = 3 7 8 9 10 11 12 13 14 15


Descritores para o Grafo( 3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 7 15

22) N* 3 Nnvlf* 9 14
```

```
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

23) N* 3 Nnvlf* 10 12
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

24) N* 3 Nnvlf* 14 15
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

25) N* 3 Nnvlf* 2 3
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

26) N* 3 Nnvlf* 14 2
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

27) N* 3 Nnvlf* 10 11
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

28) N* 3 Nnvlf* 8 7
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

29) N* 3 Nnvlf* 7 8
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

30) N* 3 Nnvlf* 4 6
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

31) N* 3 Nnvlf* 6 3
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

32) N* 3 Nnvlf* 4 5
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15



Descritores para o Grafo( 4)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 4 7 15

33) N* 4 6
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

34) N* 4 Nnvlf* 9 14
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15
```

```
35) N* 4 Nnvlf* 10 12
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

36) N* 4 Nnvlf* 14 15
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

37) N* 4 Nnvlf* 2 3
Nnvlf = 2 3 5 6 7 8 9 10 11 12 13 14 15

38) N* 4 Nnvlf* 14 2
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

39) N* 4 Nnvlf* 10 11
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

40) N* 4 Nnvlf* 8 7
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

41) N* 4 Nnvlf* 7 8
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

42) N* 4 Nnvlf* 3 4
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

43) N* 4 5
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15



Descritores para o Grafo( 5)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 6 8 11 12 15

44) N* 5 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 14 15

45) N* 5 Nnvlf* 2 3
Nnvlf = 2 3 4 6 7 9 10 14 15

46) N* 5 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 14 15

47) N* 5 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 14 15
```

```
48) N* 5 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 14 15

49) N* 5 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 14 15

50) N* 5 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 14 15

51) N* 5 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 14 15

52) N* 5 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 14 15



Descritores para o Grafo( 8)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 7 8 11 12 15

53) N* 8 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 14 15

54) N* 8 Nnvlf* 3 7
Nnvlf = 2 3 4 6 7 9 10 14 15

55) N* 8 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 14 15

56) N* 8 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 14 15

57) N* 8 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 14 15

58) N* 8 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 14 15

59) N* 8 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 14 15

60) N* 8 Nnvlf* 10 12
```

```
Nnvlf = 2 3 4 6 7 9 10 14 15

61) N* 8 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 14 15

62) N* 8 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 14 15



Descritores para o Grafo(11)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

63) N* 11 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 13 14 15

64) N* 11 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 13 14 15

65) N* 11 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 13 14 15

66) N* 11 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 13 14 15

67) N* 11 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 13 14 15

68) N* 11 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 13 14 15

69) N* 11 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 13 14 15

70) N* 11 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 13 14 15

71) N* 11 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 13 14 15



Descritores para o Grafo(12)
```

```
Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

72) N* 12 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 13 14 15

73) N* 12 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 13 14 15

74) N* 12 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 13 14 15

75) N* 12 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 13 14 15

76) N* 12 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 13 14 15

77) N* 12 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 13 14 15

78) N* 12 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 13 14 15

79) N* 12 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 13 14 15

80) N* 12 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 13 14 15


Numero Total de Descritores = 80
```

Arquivo des_pdu.tes

```
DESCRITORES PARA O CRITERIO TODOS POT-DU-CAMINHOS

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

Descritores para o Grafo( 1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 2 3 7 15

 1) N* 1 Nlf* 9 14 15
 2) N* 1 Nlf* 9 14 2
 3) N* 1 Nlf* 10 12 Nlf* 14 15
 4) N* 1 Nlf* 10 12 Nlf* 14 2
 5) N* 1 Nlf* 10 11 Nlf* 14 15
 6) N* 1 Nlf* 10 11 Nlf* 14 2
 7) N* 1 Nlf* 7 8 7
 8) N* 1 Nlf* 4 6 3
 9) N* 1 Nlf* 4 5 6 3


Descritores para o Grafo( 2)

Ni = 2 3 4 7 8 9 10 11 12 13 14 15
Nt = 2 4 7 15

10) N* 2 Nlf* 9 14 15
11) N* 2 Nlf* 9 14 2
12) N* 2 Nlf* 10 12 Nlf* 14 15
13) N* 2 Nlf* 10 12 Nlf* 14 2
14) N* 2 Nlf* 10 11 Nlf* 14 15
15) N* 2 Nlf* 10 11 Nlf* 14 2
16) N* 2 Nlf* 7 8 7
17) N* 2 3 4


Descritores para o Grafo( 3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 7 15

18) N* 3 Nlf* 9 14 15
19) N* 3 Nlf* 9 14 2 3
20) N* 3 Nlf* 10 12 Nlf* 14 15
```

21) N* 3 Nlf* 10 12 Nlf* 14 2 3
22) N* 3 Nlf* 10 11 Nlf* 14 15
23) N* 3 Nlf* 10 11 Nlf* 14 2 3
24) N* 3 7 8 7
25) N* 3 4 6 3
26) N* 3 4 5 6 3

Descritores para o Grafo(4)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 4 7 15

27) N* 4 6 Nlf* 9 14 15
28) N* 4 6 Nlf* 9 14 2 3
29) N* 4 6 Nlf* 10 12 Nlf* 14 15
30) N* 4 6 Nlf* 10 12 Nlf* 14 2 3
31) N* 4 6 Nlf* 10 11 Nlf* 14 15
32) N* 4 6 Nlf* 10 11 Nlf* 14 2 3
33) N* 4 6 Nlf* 7 8 7
34) N* 4 6 3 4
35) N* 4 5 Nlf* 9 14 15
36) N* 4 5 Nlf* 9 14 2 3
37) N* 4 5 Nlf* 10 12 Nlf* 14 15
38) N* 4 5 Nlf* 10 12 Nlf* 14 2 3
39) N* 4 5 Nlf* 10 11 Nlf* 14 15
40) N* 4 5 Nlf* 10 11 Nlf* 14 2 3
41) N* 4 5 Nlf* 7 8 7
42) N* 4 5 Nlf* 3 4

Descritores para o Grafo(5)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 6 8 11 12 15

43) N* 5 Nlf* 9 14 15
44) N* 5 Nlf* 9 14 2 3
45) N* 5 Nlf* 10 12
46) N* 5 Nlf* 10 11
47) N* 5 Nlf* 7 8
48) N* 5 Nlf* 4 6
49) N* 5 Nlf* 4 5

```
Descritores para o Grafo( 8)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 7 8 11 12 15

50) N* 8 Nlf* 9 14 15
51) N* 8 Nlf* 9 14 2 3 7
52) N* 8 Nlf* 9 14 2 Nlf* 4 6 3
53) N* 8 Nlf* 9 14 2 Nlf* 4 5
54) N* 8 Nlf* 10 12
55) N* 8 Nlf* 10 11
56) N* 8 7 8


Descritores para o Grafo(11)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

57) N* 11 Nlf* 14 15
58) N* 11 Nlf* 14 2 Nlf* 9 14
59) N* 11 Nlf* 14 2 Nlf* 10 12
60) N* 11 Nlf* 14 2 Nlf* 10 11
61) N* 11 Nlf* 14 2 Nlf* 7 8
62) N* 11 Nlf* 14 2 Nlf* 4 6 3
63) N* 11 Nlf* 14 2 Nlf* 4 5


Descritores para o Grafo(12)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

64) N* 12 Nlf* 14 15
65) N* 12 Nlf* 14 2 Nlf* 9 14
66) N* 12 Nlf* 14 2 Nlf* 10 12
67) N* 12 Nlf* 14 2 Nlf* 10 11
68) N* 12 Nlf* 14 2 Nlf* 7 8
69) N* 12 Nlf* 14 2 Nlf* 4 6 3
70) N* 12 Nlf* 14 2 Nlf* 4 5

Numero Total de Descritores = 70
```

C.2 Informações Dinâmicas

Nesta seção estão organizadas as informações dinâmicas mais relevantes geradas pela POKE-TOOL durante uma sessão de trabalho, relativa ao teste de uma unidade; no caso, da unidade ENTAB.C. Estas informações, a exemplo das informações estáticas, são armazenadas em arquivos específicos criados pela própria POKE-TOOL. O estado corrente da sessão de trabalho é mantido no arquivo POKELOG.TES. Primeiramente, são apresentadas as entradas e saídas relativas aos testes executados; estas informações são mantidas pela POKE-TOOL em arquivos denominados INPUTi.TES, TECi.TES, OUTPUTi.TES e PATHi.TES, correspondentes às entradas fornecidas como argumentos da linha de chamada, entradas via teclado, resultados obtidos e caminhos executados, respectivamente. Os arquivos relativos à análise de cobertura correspondem ao conjunto de casos de teste inicial (oito casos de teste), ou seja, ao conjunto de casos de teste elaborado inicialmente, de acordo com as diretrizes discutidas no Capítulo 6.

Arquivos INPUTi.tes

Não existem no caso da unidade ENTAB.C

Arquivos TECi.TES

Arquivo TEC1.TES

```
col 1   2   34  rest
```

Arquivo TEC2.TES

```
co
```

Arquivo TEC3.TES

```
c
```

Arquivo TEC4.TES

```
EOF[1]
```

Arquivo TEC5.TES

[1] Arquivo vazio.

NL[2]

Arquivo TEC6.TES

```
c
```

Arquivo TEC7.TES

```
col 1   2   34  rest
```

Arquivo TEC8.TES

```
 x
  x
   x
    x
     x
      x
       x
        x
         x
          x
           x
```

Arquivo OUTPUTi.TES

Arquivo OUTPUT1.TES

```
col 1 2 34 rest
```

Arquivo OUTPUT2.TES

```
co
```

Arquivo OUTPUT3.TES

```
c
```

[2]Arquivo com uma linha em branco.

Arquivo OUTPUT4.TES

Arquivo OUTPUT5.TES

Arquivo OUTPUT6.TES

```
c
```

Arquivo OUTPUT7.TES

```
col 1 2 34 rest
```

Arquivo OUTPUT8.TES

```
 x
  x
   x
x
 x
  x
   x
x
 x
  x
```

Arquivo PATHi.TES

Arquivo PATH1.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  10  12  13  14  2  3  7  9
10  12  13  14  2  3  4  5  6  3
7  9  10  12  13  14  2  3  4  6
3  4  6  3  4  5  6  3  7  9
10  12  13  14  2  3  4  6  3  4
6  3  4  5  6  3  7  9  10  12
13  14  2  3  7  9  10  12  13  14
2  3  4  6  3  4  5  6  3  7
9  10  12  13  14  2  3  7  9  10
12  13  14  2  3  7  9  10  12  13
14  2  3  7  9  10  12  13  14  2
3  7  9  14  15
```

Arquivo PATH2.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  10  12  13  14  2  3  7  9
14  15
```

Arquivo PATH3.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  14  15
```

Arquivo PATH4.TES

```
1
2  3  7  9  14  15
```

Arquivo PATH5.TES

```
1
2  3  7  9  10  11  13  14  2  3
7  9  14  15
```

Arquivo PATH6.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  10  11  13  14  2  3  7  9
14  15
```

Arquivo PATH7.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  10  12  13  14  2  3  7  9
10  12  13  14  2  3  4  5  6  3
7  9  10  12  13  14  2  3  4  6
3  4  6  3  4  5  6  3  7  9
10  12  13  14  2  3  4  6  3  4
6  3  4  5  6  3  7  9  10  12
13  14  2  3  7  9  10  12  13  14
2  3  4  6  3  4  5  6  3  7
9  10  12  13  14  2  3  7  9  10
12  13  14  2  3  7  9  10  12  13
14  2  3  7  9  10  12  13  14  2
3  7  9  10  11  13  14  2  3  7
9  14  15
```

Arquivo PATH8.TES

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

```
13  14  2  3  7  9  10  11  13  14
2  3  4  6  3  4  6  3  4  6
3  4  5  6  3  4  6  3  4  6
3  4  6  3  7  8  7  8  7  8
7  9  10  12  13  14  2  3  7  9
10  11  13  14  2  3  4  6  3  4
6  3  4  6  3  4  5  6  3  4
6  3  4  6  3  4  6  3  4  5
6  3  7  9  10  12  13  14  2  3
7  9  10  11  13  14  2  3  4  6
3  4  6  3  4  6  3  4  5  6
3  4  6  3  4  6  3  4  6  3
4  5  6  3  4  6  3  7  8  7
9  10  12  13  14  2  3  7  9  10
11  13  14  2  3  4  6  3  4  6
3  4  6  3  4  5  6  3  4  6
3  4  6  3  4  6  3  4  5  6
3  4  6  3  4  6  3  7  8  7
8  7  9  10  12  13  14  2  3  7
9  10  11  13  14  2  3  7  9  14
15
```

Arquivo Pokelog.tes

```
Nome do arquivo testado: entab.c
Numero de casos de teste: 8
Numero de avaliacoes do criterio todos-pot-du-caminhos: 8
Numero de avaliacoes do criterio todos-pot-usos: 8
Numero de avaliacoes do criterio todos-pot-usos/du: 8
Tem parametros de entrada: 0
Tem entrada pelo teclado: 1
```

Arquivo PUOUTPUT.TES

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

<1,(14,2),{ col, tabstops }>
<1,(4,5),{ col, tabstops }>
<2,(8,7),{ newcol }>
<2,(7,8),{ newcol }>
<4,(9,14),{ newcol, tabstops }>
<4,(14,15),{ newcol, tabstops }>
<4,(10,11),{ newcol, tabstops }>
<5,(14,15),{ col }>
<5,(2,3),{ col }>
<5,(14,2),{ col }>
<5,(9,14),{ col }>
<5,(10,11),{ col }>
<8,(14,15),{ col }>
<8,(3,7),{ col }>
<8,(6,3),{ col }>
<8,(4,6),{ col }>
<8,(4,5),{ col }>
<8,(14,2),{ col }>
<8,(9,14),{ col }>
<8,(10,11),{ col }>
<11,(10,12),{ col }>
<11,(10,11),{ col }>
<12,(7,8),{ col }>

Cobertura Total = 71.250000

Media da Cobertura dos Grafo(i) = 70.431721
```

Arquivo PUDUOUTPUT.TES

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS/DU nao executadas:

<1,(14,2),{ col, tabstops }>
<1,(8,7),{ tabstops }>
<1,(7,8),{ col, tabstops }>
<1,(4,5),{ col, tabstops }>
<2,(8,7),{ newcol }>
<2,(7,8),{ newcol }>
<4,(9,14),{ newcol, tabstops }>
<4,(14,15),{ newcol, tabstops }>
<4,(10,11),{ newcol, tabstops }>
<5,(14,15),{ col }>
<5,(2,3),{ col }>
<5,(14,2),{ col }>
<5,(9,14),{ col }>
<5,(10,11),{ col }>
<5,(7,8),{ col }>
<5,(4,5),{ col }>
<8,(14,15),{ col }>
<8,(3,7),{ col }>
<8,(6,3),{ col }>
<8,(4,6),{ col }>
<8,(4,5),{ col }>
<8,(14,2),{ col }>
<8,(9,14),{ col }>
<8,(10,11),{ col }>
<11,(14,15),{ col }>
<11,(10,12),{ col }>
<11,(10,11),{ col }>
<11,(7,8),{ col }>
<11,(4,5),{ col }>
<12,(14,15),{ col }>
<12,(7,8),{ col }>

Cobertura Total = 61.250000

Media da Cobertura dos Grafo(i) = 60.175312
```

Arquivo PDUOUTPUT.TES

```
POTENCIAIS-DU-CAMINHOS que nao foram executados:

Caminhos:
1 2 3 7 9 14 2
1 2 3 7 9 10 12 13 14 15
1 2 3 7 9 10 11 13 14 15
1 2 3 7 8 7
1 2 3 4 5 6 3
2 3 7 9 14 2
2 3 7 9 10 12 13 14 15
2 3 7 9 10 11 13 14 15
2 3 7 8 7
3 7 9 14 2 3
3 7 9 10 12 13 14 15
3 7 9 10 11 13 14 15
4 6 3 7 9 14 15
4 6 3 7 9 14 2 3
4 6 3 7 9 10 12 13 14 15
4 6 3 7 9 10 12 13 14 2 3
4 6 3 7 9 10 11 13 14 15
4 6 3 7 9 10 11 13 14 2 3
4 5 6 3 7 9 14 15
4 5 6 3 7 9 14 2 3
4 5 6 3 7 9 10 12 13 14 15
4 5 6 3 7 9 10 11 13 14 15
4 5 6 3 7 9 10 11 13 14 2 3
4 5 6 3 7 8 7
5 6 3 7 9 14 15
5 6 3 7 9 14 2 3
5 6 3 7 9 10 11
5 6 3 7 8
5 6 3 4 5
8 7 9 14 15
8 7 9 14 2 3 7
8 7 9 14 2 3 4 6 3
8 7 9 14 2 3 4 5
8 7 9 10 11
11 13 14 15
11 13 14 2 3 7 9 10 12
11 13 14 2 3 7 9 10 11
11 13 14 2 3 7 8
11 13 14 2 3 4 5
12 13 14 15
```

12 13 14 2 3 7 8

Cobertura Total = 41.428571

Media da Cobertura dos Grafo(i) = 42.906746

Apêndice D

Síntese dos Resultados e Principais Modelos Obtidos na Análise do Benchmark

Este apêndice apresenta uma visão geral da análise e resultados obtidos decorrentes da aplicação de um benchmark discutido no Capítulo 6, para avaliação empírica dos critérios Potenciais Usos. É apresentada uma síntese de todos os conjuntos de pontos explorados e dos respectivos modelos de regressão obtidos para o critério todos-potenciais-du-caminhos. A análise para os critérios todos-potenciais-usos e todos-potenciais-usos/du foi conduzida dentro da mesma abordagem.

D.1 Síntese da Análise do Benchmark para o Critério Todos-potenciais-du-caminhos

Na determinação dos modelos procurou-se explorar, além da influência do número de comandos de decisão na estimativa do número de casos de teste, a relação entre a ocorrência de variáveis no programa e o número de casos de teste requeridos para satisfazer os critérios baseados em análise de fluxo de dados, em particular os critérios Potenciais Usos. Um resultado secundário, do ponto de vista do objetivo principal do experimento, foi a determinação da influência dessas variáveis de controle no número de caminhos e associações não executáveis.

Um ponto importante, ressaltado no Capítulo 6, é que dois dos programas testados — RQUICK e AMATCH — são recursivos e a atual versão da POKE-TOOL não possibilita um tratamento uniforme de procedimentos recursivos e procedimentos não recursivos [MAL91c]. A análise foi então conduzida, inicialmente, para dois conjuntos de pontos distintos — um conjunto incluindo procedimentos recursivos, totalizando 29 pontos observados, e o outro excluindo os procedimentos recursivos, com 27 pontos. A

partir desses dois conjuntos de pontos, outros foram caracterizados eliminando-se as unidades (pontos) que demandaram um menor número de casos de teste: COMMAND, GETCMD, OMATCH e UNROTATE. Caracterizados os conjuntos a análise estatística foi conduzida e, em geral, as unidades (pontos observados) apontadas no Capítulo 6 caracterizaram os pontos mais influentes na determinação dos modelos para estimar as variáveis resposta *número de casos de teste* e *número de caminhos não executáveis.*

As Tabelas D.1 , D.3 , D.5 e D.7 apresentam uma síintese dos modelos obtidos para o critério todos-potenciais-du-caminhos. A primeira coluna dessas tabelas é um nome de referência ao modelo propriamente dito, ilustrado na segunda coluna sob o título Expr.; a terceira coluna (*R2*) representa o quadrado do coeficiente de correlação amostral r e indica a proporção de variabilidade explicada pelo modelo. A quarta coluna (*S*), por sua vez, indica o desvio padrão da estimativa e fornece uma indicação do desvio padrão σ. Para cada conjunto de pontos considerado foi realizada também uma análise da influência do número de potenciais du-caminhos na previsão da variável resposta número de casos de teste requeridos. Neste caso, além dos parâmetros já citados, é fornecida a estatística para teste de hipóteses referente à variável de controle.

Uma síntese das estatísticas relativas aos parâmetros dos modelos das Tabelas D.1 , D.3 , D.5 e D.7 são apresentados nas Tabelas D.2 , D.4 , D.6 e D.8. A primeira coluna dessas tabelas indica o nome de referência do modelo; da segunda à quinta coluna são indicadas as estatísticas para teste de hipóteses relativas a cada uma das variáveis de controle dos modelos explorados; as sexta e sétima colunas indicam, respectivamente, o intervalo de validade da variável de controle t do modelo e o número de pontos considerados.

A seguir são apresentados de forma mais detalhada algumas das características dos principais modelos obtidos para o critério todos-potenciais-du-caminhos para estimar as variáveis de controle *número de casos de teste* e *número de caminhos não executáveis.*

Tabela D.1: Síntese dos Modelos Obtidos Considerando o Benchmark Total

Modelo	Expr.	R2	S
M1 (txct)	ct = 1.68 + 1.79 t	25.2	9.851
M2 (txct)	ct = 1.67 + 1.78 t + 0.003 variav	25.2	10.04
M3 (txct)	ct = 1.04 + 1.44 t + 0.223 def	27.4	9.890
M4 (txct)	ct = 0.44 + 1.25 t + 0.668 nodef	28.9	9.786
M5 (txctr1)	ct = - 18.8 + 5.16 t	74.9	6.049
M6 (txctr1)	ct = - 19.4 + 5.04 t + 0.216 variav	75.2	6.143
M7 (txctr1)	ct = - 18.9 + 4.87 t + 0.152 def	75.5	6.102
M8 (txctr1)	ct = - 18.9 + 4.94 t + 0.212 nodef	75.2	6.140
M9 (txctr11)	ct = - 17.1 + 4.82 t	75.1	5.548
M10 (txctr11)	ct = - 17.9 + 4.66 t + 0.271 variav	75.7	5.608
M11 (txctr11)	ct = - 17.2 + 4.46 t + 0.180 def	76.2	5.549
M12 (txctr11)	ct = - 17.2 + 4.60 t + 0.201 nodef	75.5	5.633
M13 (txctr12)	ct = - 16.4 + 4.75 t	64.4	6.049
M14 (txctr12)	ct = - 17.0 + 4.56 t + 0.271 variav	65.0	6.127
M15 (txctr12)	ct = - 16.4 + 4.41 t + 0.165 def	65.4	6.091
M16 (txctr12)	ct = - 16.4 + 4.47 t + 0.243 nodef	65.0	6.132
M17 (txctr13)	ct = - 13.0 + 4.11 t	61.9	5.320
M18 (txctr13)	ct = - 13.7 + 3.84 t + 0.367 variav	63.7	5.317
M19 (txctr13)	ct = - 12.9 + 3.67 t + 0.206 def	64.4	5.269
M20 (txctr13)	ct = - 13.0 + 3.83 t + 0.247 nodef	62.9	5.379

Modelo	Expr.	C12	R2	S
MI1 (txct)	ct = 6.30 + 0.0985 ducam	10.13	79.2	5.201
MI2 (txctr1)	ct = 6.13 + 0.0995 ducam	9.73	79.8	5.429
MI3 (txctr11	ct = 6.03 + 0.102 ducam	8.73	75.3	5.531
MI4 (txctr12	ct = 6.58 + 0.0894 ducam	8.17	74.4	5.131
MI5 (txctr13	ct = 6.73 + 0.0862 ducam	6.15	63.3	5.230

Tabela D.2: Síntese de Parâmetros dos Modelos da Tabela 1

Modelo	C1	C2	C3	C4	t	Num.Pontos
M1	3.02				3-15	29
M2	2.71	0.01			3-15	29
M3	2.02		0.89		3-15	29
M4	1.68			1.17	3-15	29
M5	8.47				3-12	26
M6	7.64	0.52			3-12	26
M7	6.75		0.76		3-12	26
M8	6.62			0.54	3-12	26
M9	8.33				3-12	25
M10	7.48	0.71			3-12	25
M11	6.59		0.99		3-12	25
M12	6.59			0.56	3-12	25
M13	6.45				3-10	25
M14	5.72	0.65			3-10	25
M15	5.22		0.83		3-10	25
M16	5.13	0.62			3-10	25
M17	5.98				3-10	24
M18	5.22	1.01			3-10	24
M19	4.74		1.19		3-10	24
M20	4.79			0.72	3-10	24

Tabela D.3: Síntese dos Modelos Obtidos Excluída a Unidade UNROTATE

Modelo	Expr.	R2	S
M1 (txctv)	ct = 2.06 + 1.77 t	25.5	9.905
M2 (txctv)	ct = 2.10 + 1.79 t - 0.021 variav	25.5	10.10
M3 (txctv)	ct = 1.37 + 1.35 t + 0.272 def	28.7	9.879
M4 (txctv)	ct = 0.57 + 1.02 t + 0.931 nodef	32.1	9.644
M5 (txctv1)	ct = - 18.7 + 5.22 t	78.2	5.691
M6 (txctv1)	ct = - 19.1 + 5.15 t + 0.124 variav	78.3	5.806
M7 (txctv1)	ct = - 18.9 + 4.75 t + 0.251 def	79.8	5.595
M8 (txctv1)	ct = - 18.8 + 4.67 t + 0.531 nodef	80.0	5.575
M9 (txctv11)	ct = - 17.1 + 4.88 t	78.8	5.179
M10 (txctv11)	ct = - 17.6 + 4.78 t + 0.182 variav	79.0	5.269
M11 (txctv11)	ct = - 17.2 + 4.36 t + 0.275 def	81.2	4.990
M12 (txctv11)	ct = - 17.2 + 4.37 t + 0.503 nodef	80.7	5.050
M13 (txctv12)	ct = - 16.7 + 4.86 t	68.8	5.716
M14 (txctv12)	ct = - 17.0 + 4.74 t + 0.174 variav	69.1	5.824
M15 (txctv12)	ct = - 16.7 + 4.35 t + 0.260 def	71.4	5.600
M16 (txctv12)	ct = - 16.6 + 4.26 t + 0.551 nodef	71.6	5.577
M17 (txctv13)	ct = - 13.4 + 4.25 t	67.2	4.987
M18 (txctv13)	ct = - 13.9 + 4.04 t + 0.276 variav	68.2	5.031
M19 (txctv13)	ct = - 13.4 + 3.65 t + 0.293 def	72.0	4.724
M20 (txctv13)	ct = - 13.4 + 3.68 t + 0.530 nodef	71.0	4.805

Modelo	Expr.	C12	R2	S
MI1 (txctv)	ct = 6.60 + 0.0981 ducam	10.27	80.2	5.101
MI2 (txctv1)	ct = 6.46 + 0.0990 ducam	9.85	80.8	5.336
MI3 (txctv11)	ct = 6.35 + 0.101 ducam	8.49	76.6	5.437
MI4 (txctv12)	ct = 6.88 + 0.0891 ducam	8.30	75.8	5.031
MI5 (txctv13)	ct = 7.01 + 0.0864 ducam	6.27	65.2	5.137

Tabela D.4: Síntese de Parâmetros dos Modelos da Tabela 3

Modelo	C1	C2	C3	C4	t	Num.Pontos
M1	2.98				3-15	28
M2	2.70	-0.05			3-15	28
M3	1.88		1.06		3-15	28
M4	1.35			1.56	3-15	28
M5	9.08				3-12	25
M6	8.22	0.31			3-12	25
M7	7.15		1.34		3-12	25
M8	6.81			1.40	3-12	25
M9	9.04				3-12	24
M10	8.12	0.51			3-12	24
M11	7.14		1.64		3-12	24
M12	6.91			1.46	3-12	24
M13	6.96				3-10	24
M14	6.21	0.44			3-10	24
M15	5.6		1.38		3-10	24
M16	5.35			1.45	3-10	24
M17	6.56				3-10	23
M18	5.73	0.80			3-10	23
M19	5.25		1.84		3-10	23
M20	5.14			1.62	3-10	23

Tabela D.5: Síntese dos Modelos Obtidos Excluídas as Unidades Recursivas

Modelo	Expr.	R2	S
M1 (t-ct)	ct = 2.48 + 1.73 t	24.7	10.01
M2 (t-ct)	ct = 2.32 + 1.67 t + 0.088 variav	24.9	10.21
M3 (t-ct)	ct = 1.88 + 1.35 t + 0.243 def	27.4	10.04
M4 (t-ct)	ct = 1.30 + 1.16 t + 0.706 nodef	28.8	9.942
M5 (t-ctr11)	ct = - 16.4 + 4.79 t	77.8	5.337
M6 (t-ctr11)	ct = - 17.7 + 4.46 t + 0.546 variav	80.0	5.198
M7 (t-ctr11)	ct = - 16.5 + 4.36 t + 0.222 def	79.5	5.260
M8 (t-ctr11)	ct = - 16.4 + 4.53 t + 0.251 nodef	78.3	5.404
M9 (t-ctr12)	ct = - 15.7 + 4.73 t	67.4	5.337
M10 (t-ctr12)	ct = - 16.8 + 4.34 t + 0.556 variav	70.0	5.198
M11 (t-ctr12)	ct = - 15.7 + 4.32 t + 0.208 def	69.2	5.260
M12 (t-ctr12)	ct = - 15.7 + 4.41 t + 0.297 nodef	68.3	5.404
M13 (t-ctr13)	ct = - 12.5 + 4.11 t	65.7	5.102
M14 (t-ctr13)	ct = - 13.5 + 3.61 t + 0.656 variav	71.1	4.807
M15 (t-ctr13)	ct = - 12.4 + 3.60 t + 0.242 def	69.2	4.960
M16 (t-ctr13)	ct = - 12.5 + 3.80 t + 0.282 nodef	66.9	5.144

Modelo	Expr.	C12	R2	S
MI1 (t-ct.res)	ct = 6.79 + 0.0973 ducam	10.27	80.8	5.053
MI2 (t-ctr11)	ct = 6.56 + 0.101 ducam	8.45	77.3	5.403
MI3 (t-ctr12)	ct = 7.08 + 0.0882 ducam	8.31	76.7	5.403
MI4 (t-ctr13)	ct = 7.22 + 0.0852 ducam	6.25	66.2	5.071

Tabela D.6: Síntese de Parâmetros dos Modelos da Tabela 5

Modelo	C1	C2	C3	C4	t	Num.Pontos
M1	2.87				3-15	27
M2	2.46	0.21			3-15	27
M3	1.85		0.94		3-15	27
M4	1.51			1.17	3-15	27
M5	8.58				3-12	23
M6	7.59	1.46			3-12	23
M7	6.76		1.27		3-12	23
M8	6.72			0.70	3-12	23
M9	6.59				3-10	23
M10	5.68	1.33			3-10	23
M11	5.31		1.07		3-10	23
M12	5.21			0.75	3-10	23
M13	6.20				3-10	22
M14	5.32	1.88			3-10	22
M15	4.93		1.47		3-10	22
M16	4.97			0.82	3-10	22

Tabela D.7: Síntese dos Modelos Excluídas as Unidades Recursivas e a Unidade UNROTATE

Modelo	Expr.	R2	S
M1 (t-ctv)	ct = 2.92 + 1.72 t	25.0	10.06
M2 (t-ctv)	ct = 2.80 + 1.67 t + 0.064 variav	25.1	10.27
M3 (t-ctv)	ct = 2.29 + 1.24 t + 0.299 def	29.0	10.00
M4 (t-ctv)	ct = 1.50 + 0.885 t + 1.02 nodef	32.6	9.743
M5 (t-ctv1)	ct = - 17.9 + 5.18 t	81.1	5.389
M6 (t-ctv1)	ct = - 18.9 + 4.94 t + 0.397 variav	82.1	5.383
M7 (t-ctv1)	ct = - 18.0 + 4.60 t + 0.315 def	83.7	5.123
M8 (t-ctv1)	ct = - 18.0 + 4.50 t + 0.669 nodef	83.8	5.113
M9 (t-ctv11)	ct = - 16.3 + 4.86 t	82.2	4.837
M10 (t-ctv11)	ct = - 17.4 + 4.58 t + 0.454 variav	83.7	4.750
M11 (t-ctv11)	ct = - 16.4 + 4.23 t + 0.333 def	85.8	4.433
M12 (t-ctv11)	ct = - 16.5 + 4.24 t + 0.624 nodef	85.0	4.545
M13 (t-ctv12)	ct = - 16.0 + 4.86 t	72.7	5.716
M14 (t-ctv12)	ct = - 16.8 + 4.53 t + 0.454 variav	74.5	5.824
M15 (t-ctv12)	ct = - 16.1 + 4.24 t + 0.320 def	76.7	5.600
M16 (t-ctv12)	ct = - 16.0 + 4.15 t + 0.678 nodef	76.8	5.577
M17 (t-ctv13)	ct = - 12.9 + 4.26 t	72.2	4.639
M18 (t-ctv13)	ct = - 13.7 + 3.81 t + 0.560 variav	76.2	4.412
M19 (t-ctv13)	ct = - 12.8 + 3.57 t + 0.344 def	79.1	4.133
M20 (t-ctv13)	ct = - 13.0 + 3.62 t + 0.633 nodef	77.5	4.290

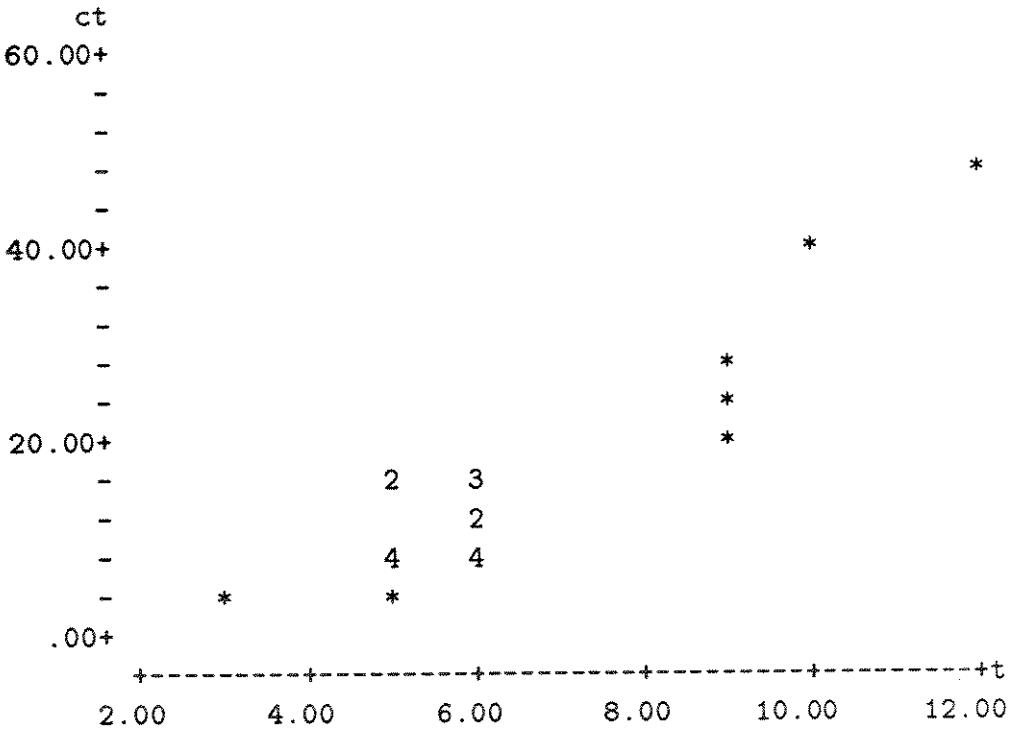
Modelo	Expr.	C12	R2	S
MD1 (t-ctv)	ct = 7.12 + 0.0968 ducam	10.51	82.1	4.909
MD2 (t-ctv1)	ct = 7.04 + 0.0976 ducam	10.03	82.7	5.155
MD3 (t-ctv11)	ct = 6.93 + 0.100 ducam	8.65	78.9	5.257
MD4 (t-ctv12)	ct = 7.44 + 0.0879 ducam	8.55	78.5	5.031
MD5 (t-ctv13)	ct = 7.55 + 0.0854 ducam	6.46	68.7	4.92

Tabela D.8: Síntese de Parâmetros dos Modelos da Tabela 7

Modelo	C1	C2	C3	C4	t	Num.Pontos
M1	2.83				3-15	26
M2	2.45	0.15			3-15	26
M3	1.70		1.14		3-15	26
M4	1.13			1.61	3-15	26
M5	9.50				3-12	23
M6	8.38	1.02			3-12	23
M7	7.52		1.80		3-12	23
M8	7.08			1.82	3-12	23
M9	9.60				3-12	22
M10	8.49	1.32			3-12	22
M11	7.77		2.20		3-12	22
M12	7.38			1.91	3-12	22
M13	7.29				3-10	22
M14	6.30	1.16			3-10	22
M15	5.95		1.82		3-10	22
M16	5.65			1.84	3-10	22
M17	7.02				3-10	21
M18	6.05	1.73			3-10	21
M19	5.88		2.44		3-10	21
M20	5.65			2.05	3-10	21

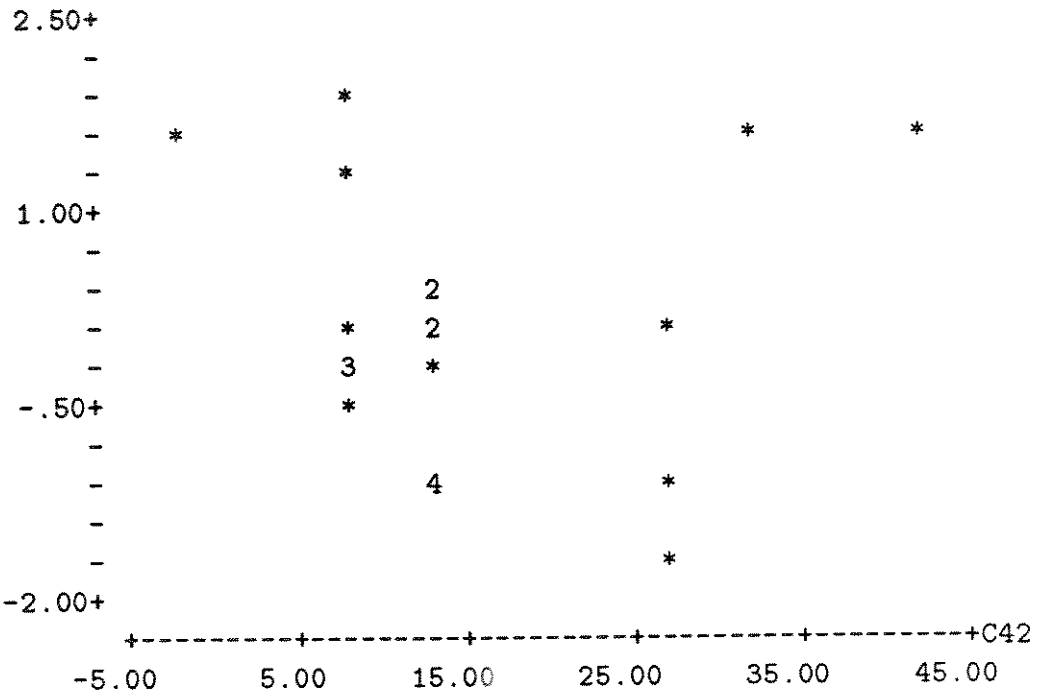
D.1.1 Modelos de Regressão: número de casos de teste.

```
MTB > choose 0 12 in c1,corr c2-c4,c11,c12,c23,put c1-c4,c11,c12,c23
MTB > omit  20 in c11,corr c1-c4,c12,c23,put c11,c1-c4,c12,c23
MTB > omit 40  in c23,corr c1-c4,c11,c12,put c23,c1-c4,c11,c12
MTB > plot c23 c1
```

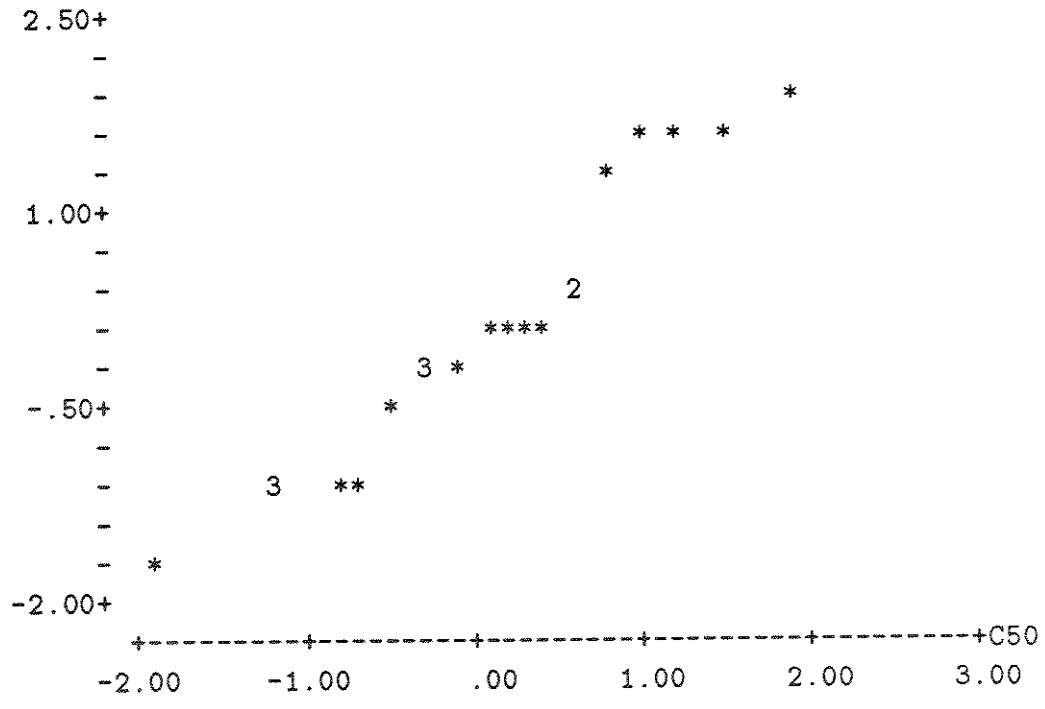


```
MTB > regress c23 1 c1,c41,c42

THE REGRESSION EQUATION IS
ct = - 16.3 + 4.86 t

                                ST. DEV.     T-RATIO =
COLUMN      COEFFICIENT         OF COEF.     COEF/S.D.
                -16.318           3.404         -4.79
t                4.8581           0.5062         9.60

S = 4.837

R-SQUARED = 82.2 PERCENT
R-SQUARED = 81.3 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

 DUE TO      DF           SS       MS=SS/DF
REGRESSION    1       2155.2         2155.2
RESIDUAL     20        468.0           23.4
TOTAL        21       2623.3

                        Y    PRED. Y   ST.DEV.
 ROW         t         ct      VALUE   PRED. Y  RESIDUAL    ST.RES.
   9      12.0      48.00      41.98      3.01      6.02       1.59 X

X DENOTES AN OBS. WHOSE X VALUE GIVES IT LARGE INFLUENCE.

DURBIN-WATSON STATISTIC = 1.18
```

```
MTB > plot c41 c42
```




```
MTB > nscores c41 c50
MTB > plot c41 c50
```




```
MTB > regress c23 2 c1 c2,c41,c42
```

THE REGRESSION EQUATION IS
ct = - 17.4 + 4.58 t + 0.454 variav

COLUMN	COEFFICIENT	ST. DEV. OF COEF.	T-RATIO = COEF/S.D.
	-17.431	3.447	-5.06
t	4.5812	0.5395	8.49
variav	0.4539	0.3436	1.32

S = 4.750

R-SQUARED = 83.7 PERCENT
R-SQUARED = 81.9 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

DUE TO	DF	SS	MS=SS/DF
REGRESSION	2	2194.6	1097.3
RESIDUAL	19	428.7	22.6
TOTAL	21	2623.3	

FURTHER ANALYSIS OF VARIANCE
SS EXPLAINED BY EACH VARIABLE WHEN ENTERED IN THE ORDER GIVEN

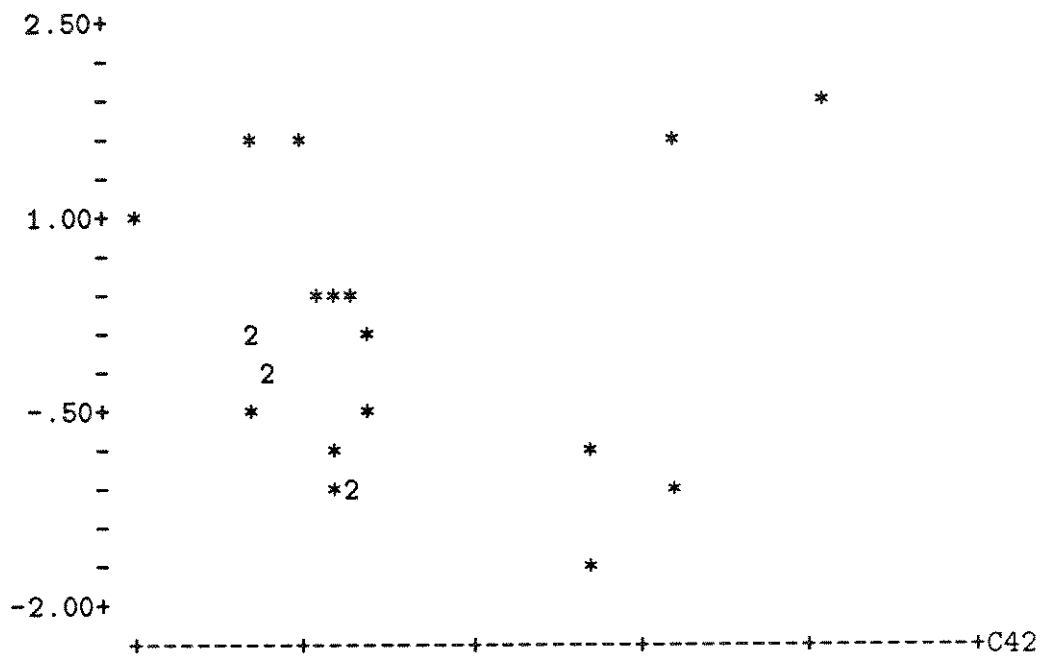
DUE TO	DF	SS
REGRESSION	2	2194.6
t	1	2155.2
variav	1	39.4

ROW	t	Y ct	PRED. Y VALUE	ST.DEV. PRED. Y	RESIDUAL	ST.RES.
18	9.0	28.00	31.51	3.52	-3.51	-1.10 X

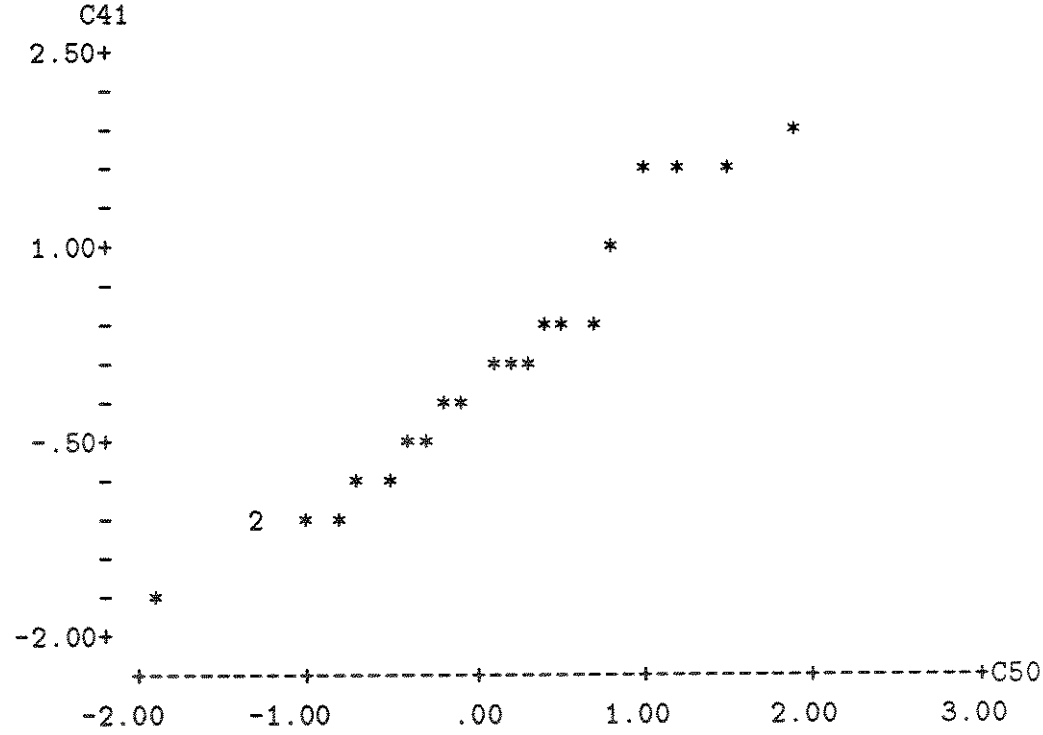
X DENOTES AN OBS. WHOSE X VALUE GIVES IT LARGE INFLUENCE.

DURBIN-WATSON STATISTIC = 1.07

```
MTB > plot c41 c42
```




```
MTB > nscores c41 c50
MTB > plot c41 c50
```




```
MTB > regress c23 2 c1 c3,c41,c42

THE REGRESSION EQUATION IS
```

ct = - 16.4 + 4.23 t + 0.333 def

COLUMN	COEFFICIENT	ST. DEV. OF COEF.	T-RATIO = COEF/S.D.
	-16.448	3.120	-5.27
t	4.2320	0.5446	7.77
def	0.3326	0.1515	2.20

S = 4.433

R-SQUARED = 85.8 PERCENT
R-SQUARED = 84.3 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

DUE TO	DF	SS	MS=SS/DF
REGRESSION	2	2249.9	1125.0
RESIDUAL	19	373.3	19.6
TOTAL	21	2623.3	

FURTHER ANALYSIS OF VARIANCE
SS EXPLAINED BY EACH VARIABLE WHEN ENTERED IN THE ORDER GIVEN

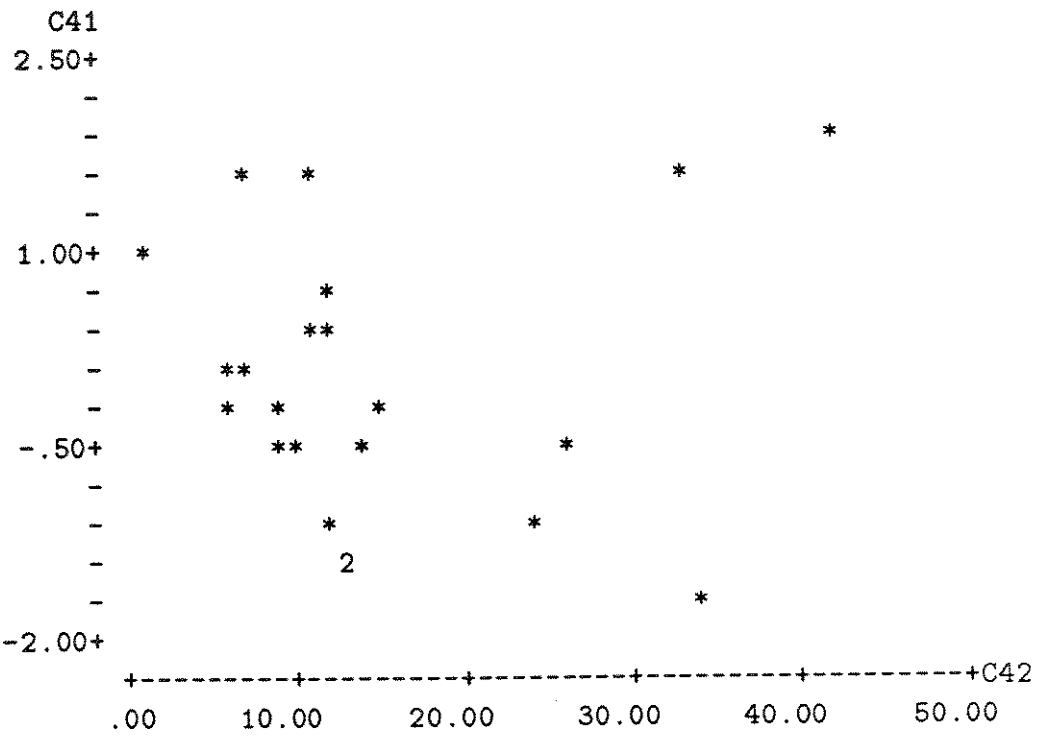
DUE TO	DF	SS
REGRESSION	2	2249.9
t	1	2155.2
def	1	94.7

ROW	t	Y ct	PRED. Y VALUE	ST.DEV. PRED. Y	RESIDUAL	ST.RES.
18	9.0	28.000	33.614	3.215	-5.614	-1.84 X

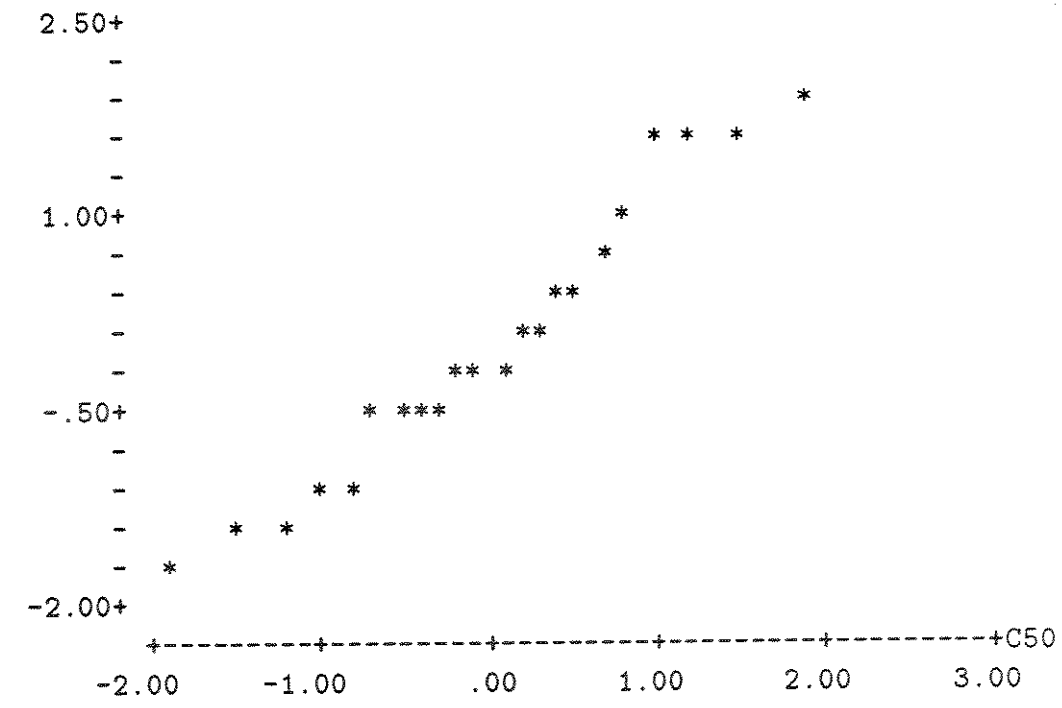
X DENOTES AN OBS. WHOSE X VALUE GIVES IT LARGE INFLUENCE.

DURBIN-WATSON STATISTIC = 1.13

```
MTB > plot c41 c42
```




```
MTB > nscores c41 c50
MTB > plot c41 c50
```




```
MTB > regress c23 2 c1 c4,c41,c42

THE REGRESSION EQUATION IS
```

ct = - 16.5 + 4.24 t + 0.624 nodef

COLUMN	COEFFICIENT	ST. DEV. OF COEF.	T-RATIO = COEF/S.D.
	-16.506	3.200	-5.16
t	4.2412	0.5748	7.38
nodef	0.6241	0.3264	1.91

S = 4.545

R-SQUARED = 85.0 PERCENT
R-SQUARED = 83.5 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

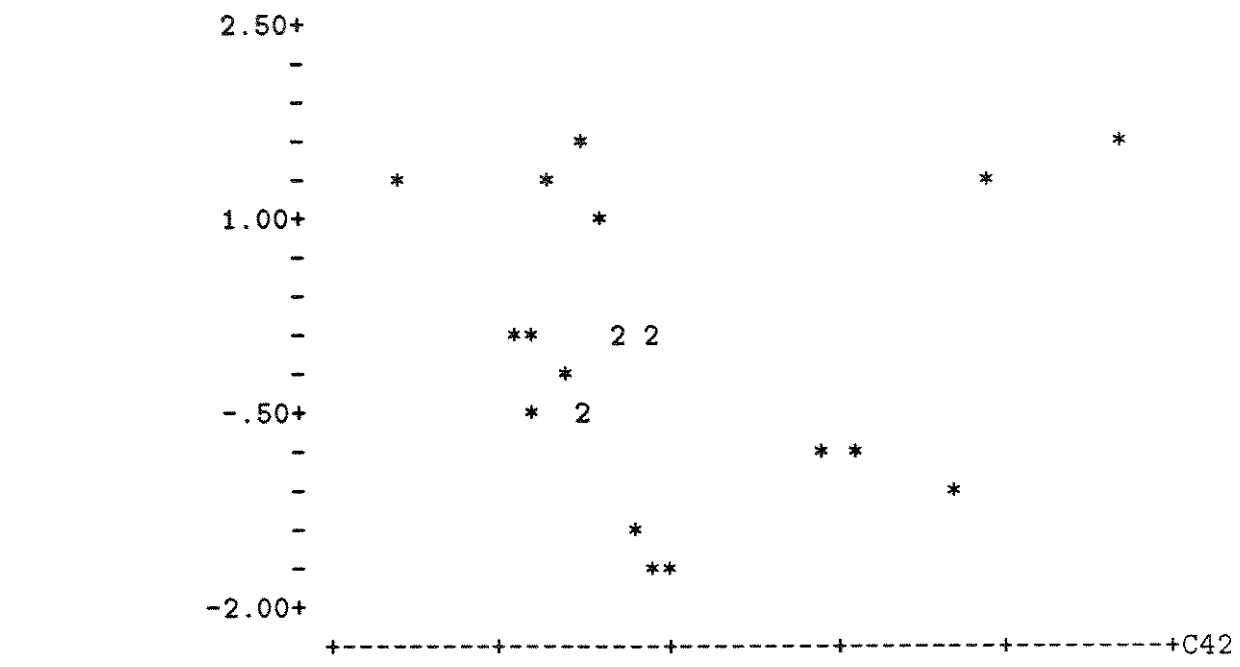
DUE TO	DF	SS	MS=SS/DF
REGRESSION	2	2230.8	1115.4
RESIDUAL	19	392.5	20.7
TOTAL	21	2623.3	

FURTHER ANALYSIS OF VARIANCE
SS EXPLAINED BY EACH VARIABLE WHEN ENTERED IN THE ORDER GIVEN

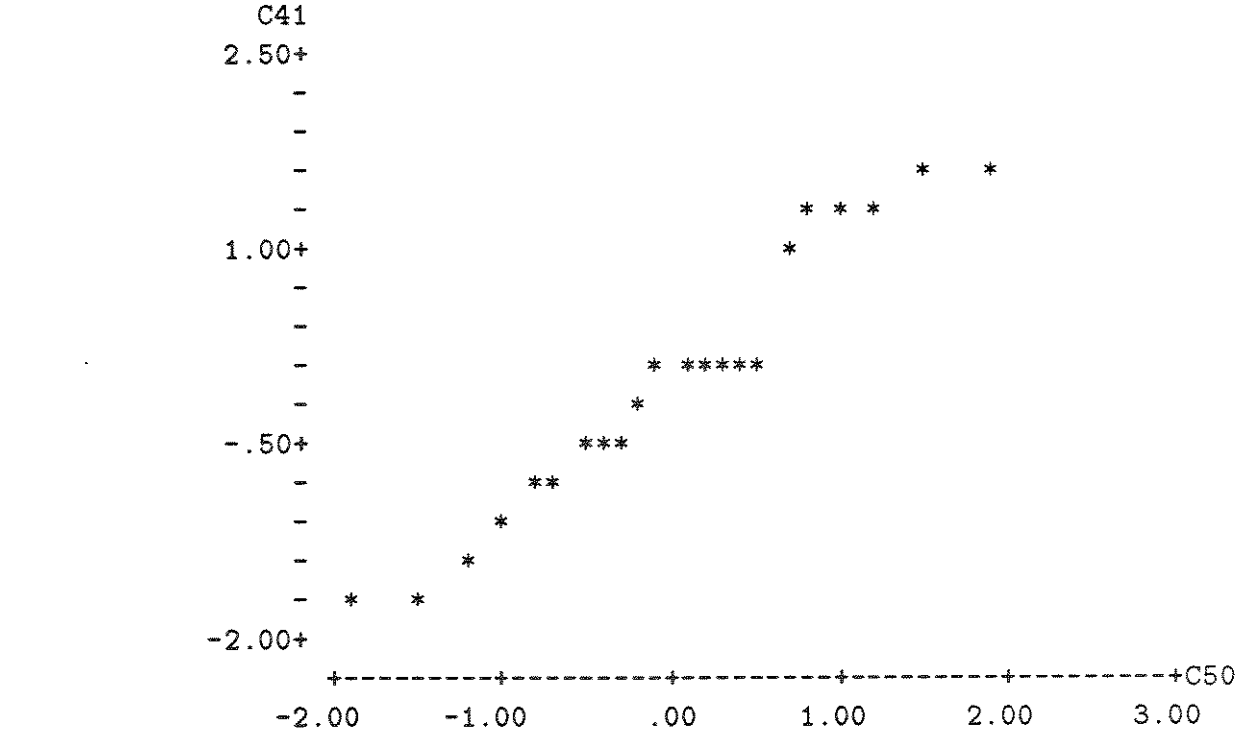
DUE TO	DF	SS
REGRESSION	2	2230.8
t	1	2155.2
nodef	1	75.5

DURBIN-WATSON STATISTIC = 1.22

```
MTB > plot c41 c42
```

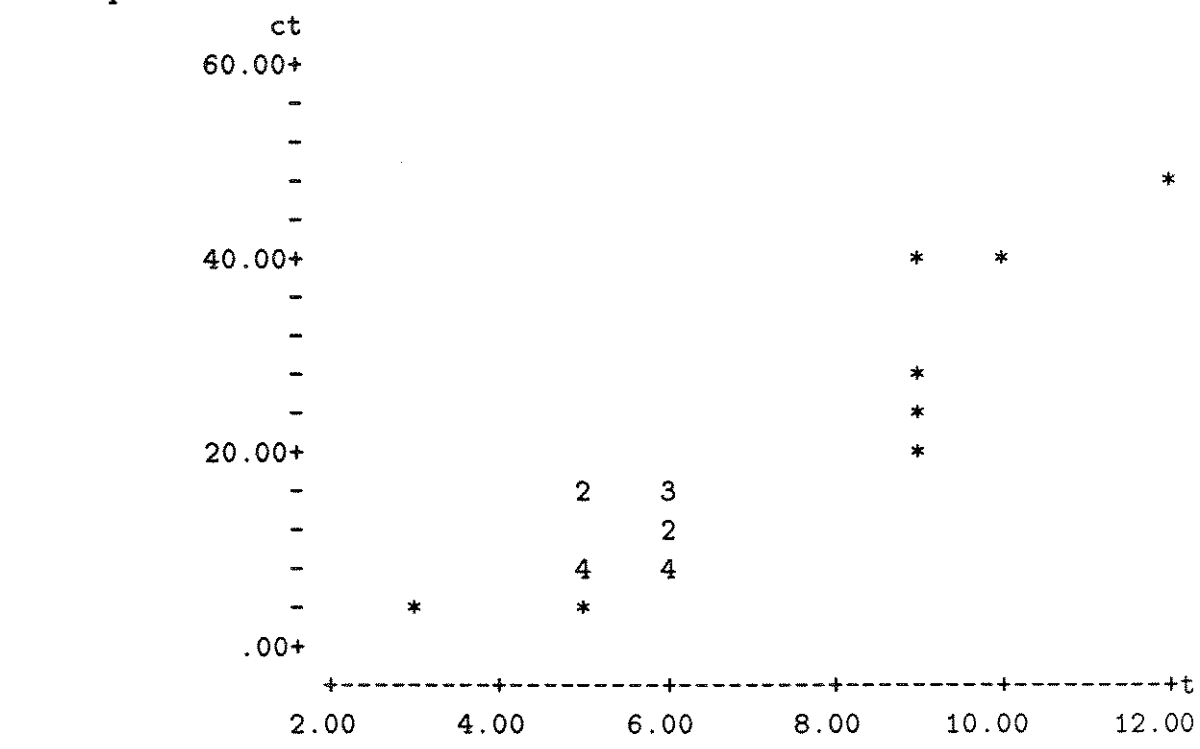



```
MTB > nscores c41 c50
MTB > plot c41 c50
```



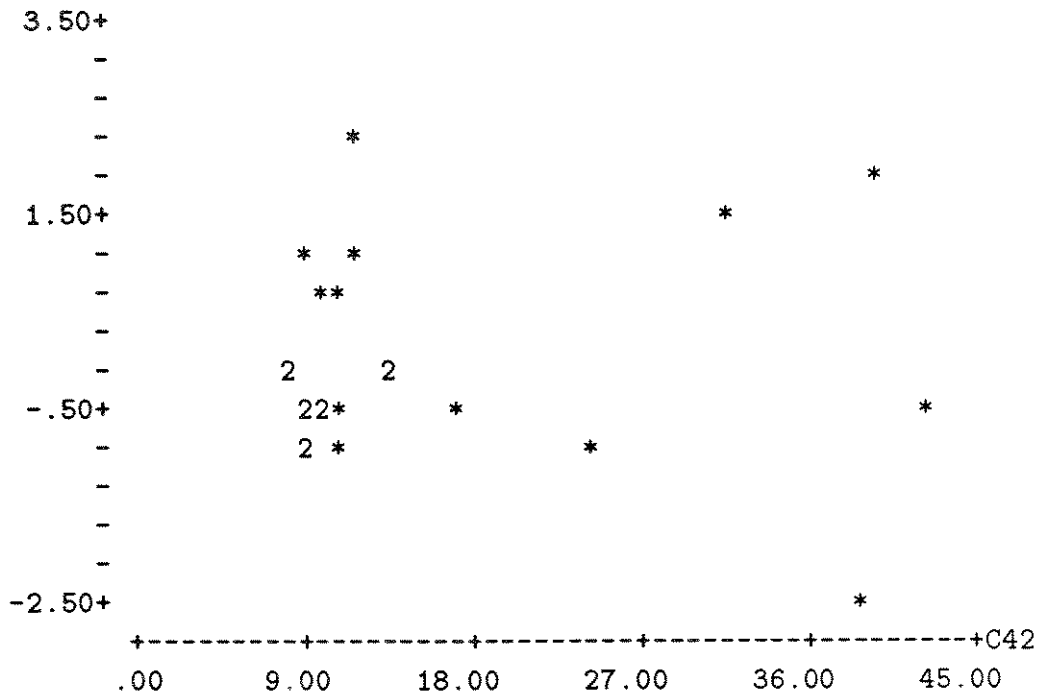

```
MTB > choose 0 12 in c1,corr c2-c4,c11,c12,c23,put c1-c4,c11,c12,c23
MTB > omit 20 in c11,corr c1-c4,c12,c23,put c11,c1-c4,c12,c23
```

```
MTB > print c1 c2 c3 c4 c11 c12 c23
MTB > plot c23 c1
```

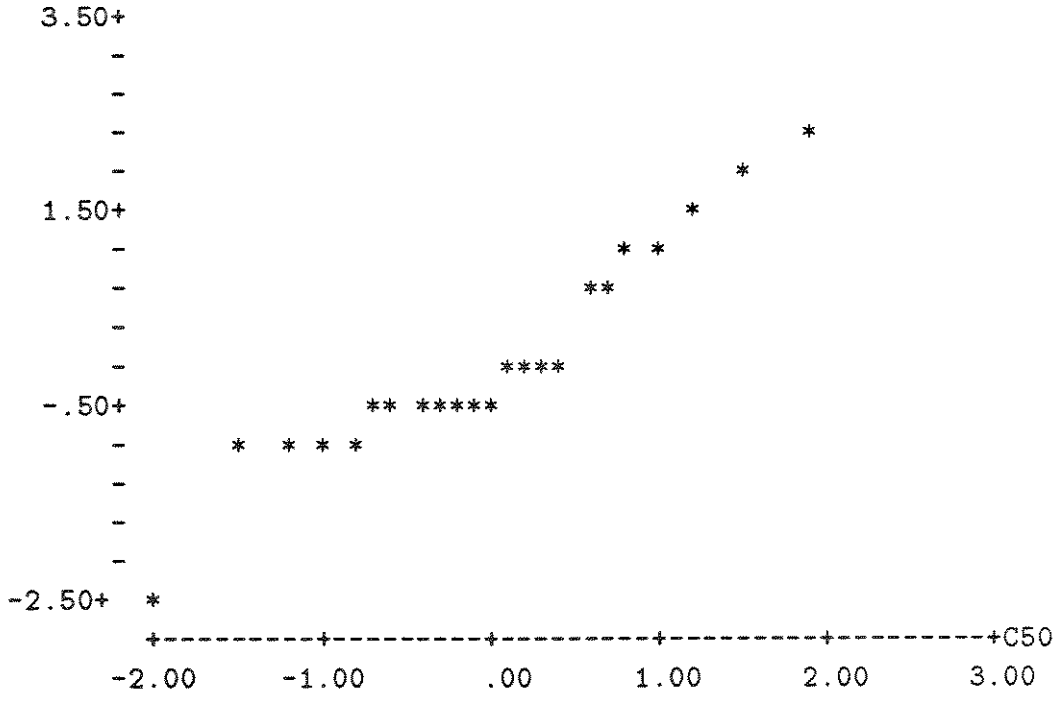


```
MTB > regress c23 1 c12,c41,c42

THE REGRESSION EQUATION IS
ct = 7.04 + 0.0976 ducam

                                ST. DEV.    T-RATIO =
COLUMN      COEFFICIENT        OF COEF.    COEF/S.D.
                 7.044           1.392         5.06
ducam         0.097647        0.009737        10.03

S = 5.155

R-SQUARED = 82.7 PERCENT
R-SQUARED = 81.9 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

 DUE TO      DF           SS       MS=SS/DF
REGRESSION    1       2671.9         2671.9
RESIDUAL     21        558.0           26.6
TOTAL        22       3229.8

                       Y   PRED. Y   ST.DEV.
 ROW    ducam         ct     VALUE   PRED. Y  RESIDUAL    ST.RES.
  11       49      23.00     11.83      1.15     11.17      2.22R
  18      322      28.00     38.49      2.49    -10.49     -2.32R
  19      357      40.00     41.90      2.81     -1.90     -0.44 X

R DENOTES AN OBS. WITH A LARGE ST. RES.
X DENOTES AN OBS. WHOSE X VALUE GIVES IT LARGE INFLUENCE.

DURBIN-WATSON STATISTIC = 1.91
```

```
MTB > plot c41 c42
```

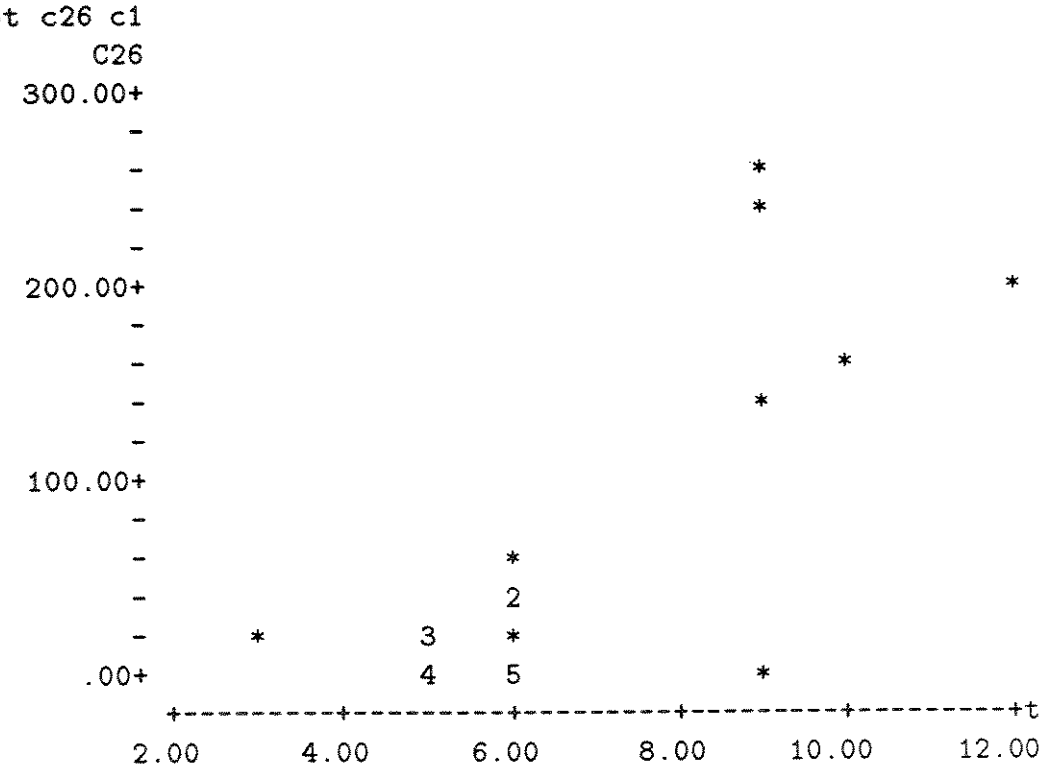



```
MTB > nscores c41 c50
MTB > plot c41 c50
```



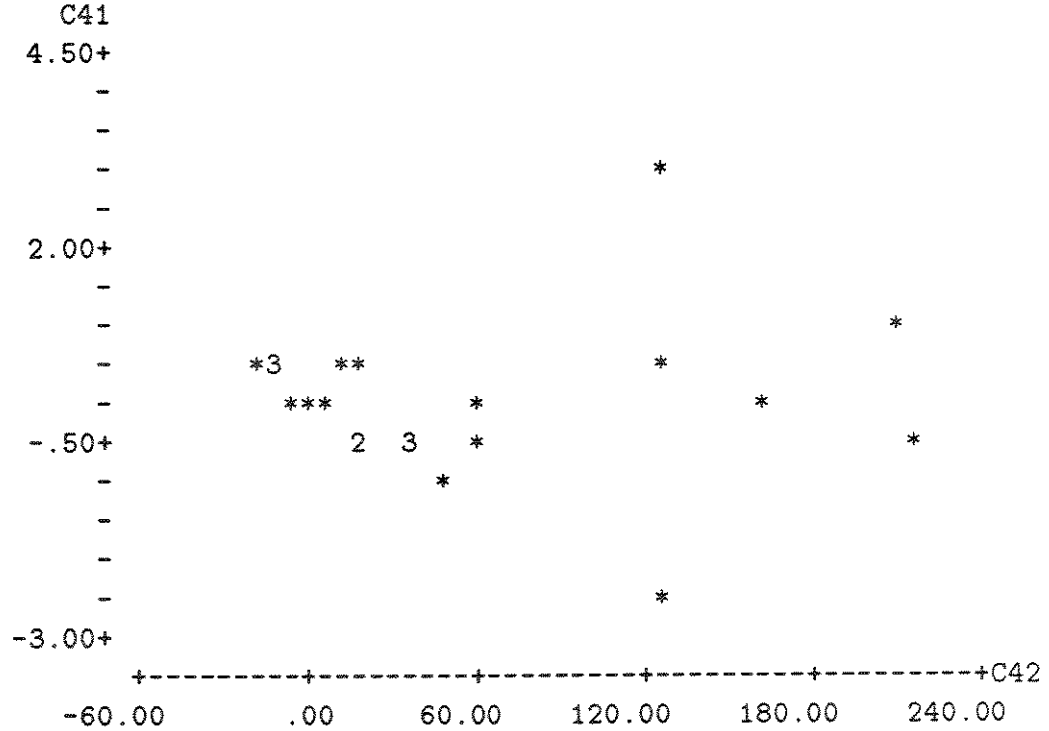
D.1.2 Modelos de Regressão: número de caminhos não executáveis.

```
MTB > omit 2 in c23, corr. c1-c4,c11,c12,c26,put c23,c1-c4,c11,c12,c26
MTB > omit 87 in c12, corr. c1-c4,c11,c26,c23,put c12,c1-c4,c11,c26,c23
MTB > choose 0 12 in c1,corr c2-c4,c11,c12,c26,c23,put c1-c4,c11,c12,c26,c23
MTB > omit 20 in c11,corr c1-c4,c12,c26,c23,put c11,c1-c4,c12,c26,c23
MTB > print c1 c2 c3 c4 c11 c12 c26

MTB > plot c26 c1
```



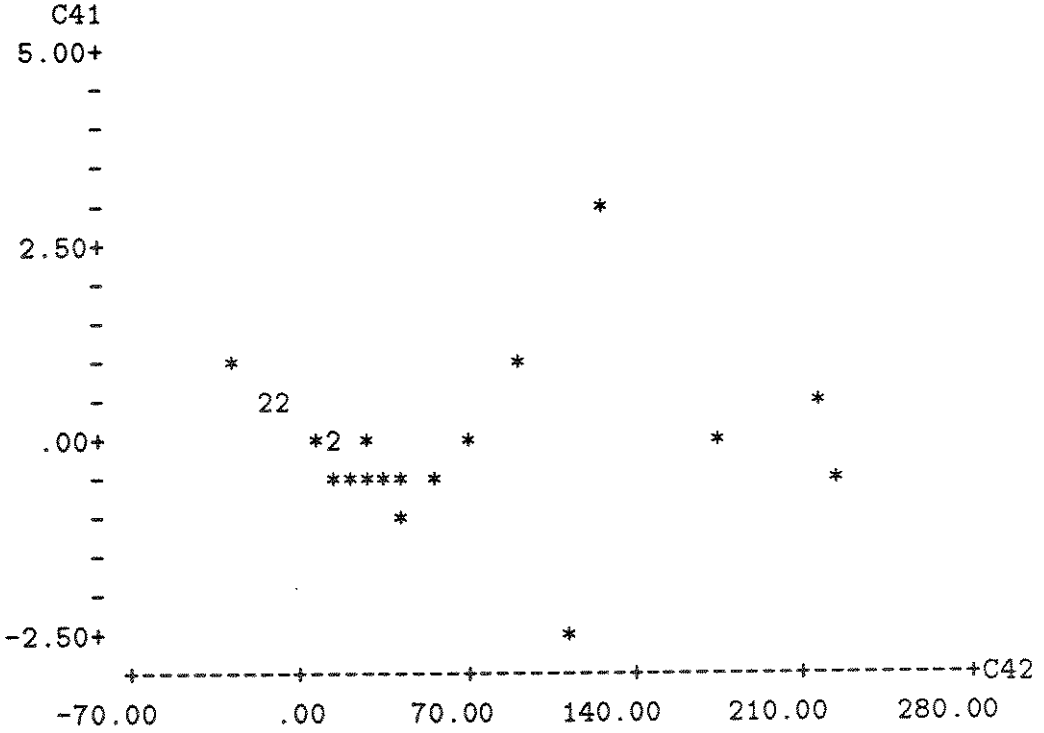
```
MTB > regress c26 1 c1,c41,c42

THE REGRESSION EQUATION IS
C26 = - 152 + 31.6 t

                                ST. DEV.    T-RATIO =
COLUMN      COEFFICIENT         OF COEF.    COEF/S.D.
                -151.78           35.43       -4.28
t                31.633           5.180        6.11

S = 51.21

R-SQUARED = 64.0 PERCENT
R-SQUARED = 62.3 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

 DUE TO      DF           SS       MS=SS/DF
REGRESSION    1        97800          97800
RESIDUAL     21        55066           2622
TOTAL        22       152866

DURBIN-WATSON STATISTIC = 1.69
MTB > plot c41 c42
              C41
            3.00+
                -                                *
                -                                *
                -
                -    *
            1.00+
                -              *
                -         4    *          *    *
                -         3    *
                -              6                    *
           -1.00+
                -
                -
                -
                -                                *
           -3.00+
                  +---------+---------+---------+---------+---------+C42
               -80.00       .00      80.00     160.00    240.00    320.00

MTB > regress c26 2 c1 c2,c41,c42
```

```
THE REGRESSION EQUATION IS
C26 = - 173 + 26.7 t + 8.37 variav

                                ST. DEV.     T-RATIO =
COLUMN      COEFFICIENT         OF COEF.     COEF/S.D.
              -173.28             32.67        -5.30
t              26.723              5.002        5.34
variav          8.373              3.293        2.54

S = 45.61

R-SQUARED = 72.8 PERCENT
R-SQUARED = 70.1 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

 DUE TO      DF           SS       MS=SS/DF
REGRESSION    2       111252          55626
RESIDUAL     20        41614           2081
TOTAL        22       152866

DURBIN-WATSON STATISTIC = 1.99
MTB > plot c41 c42
```



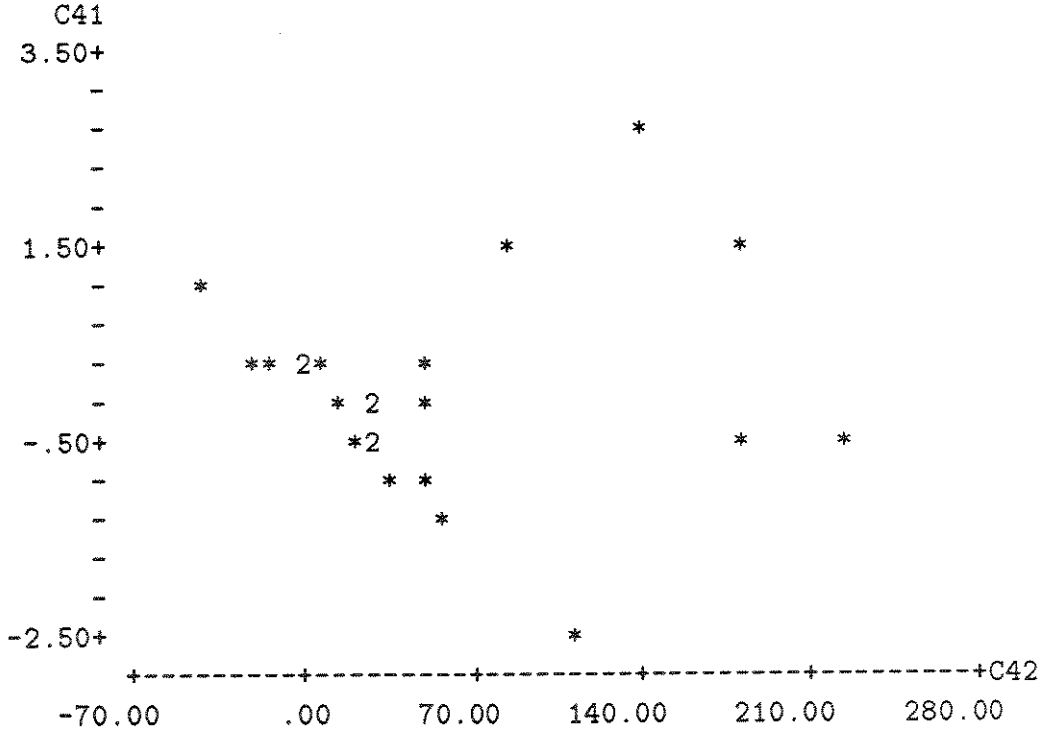

```
MTB > regress c26 2 c1 c3,c41,c42
```

```
THE REGRESSION EQUATION IS
C26 = - 154 + 23.6 t + 4.33 def

                                ST. DEV.     T-RATIO =
COLUMN       COEFFICIENT        OF COEF.     COEF/S.D.
                -154.17           30.55         -5.05
t                23.621           5.263          4.49
def               4.334           1.507          2.88

S = 44.14

R-SQUARED = 74.5 PERCENT
R-SQUARED = 72.0 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

 DUE TO      DF           SS       MS=SS/DF
REGRESSION    2       113903          56952
RESIDUAL     20        38963           1948
TOTAL        22       152866

DURBIN-WATSON STATISTIC = 2.26
MTB > plot c41 c42
```

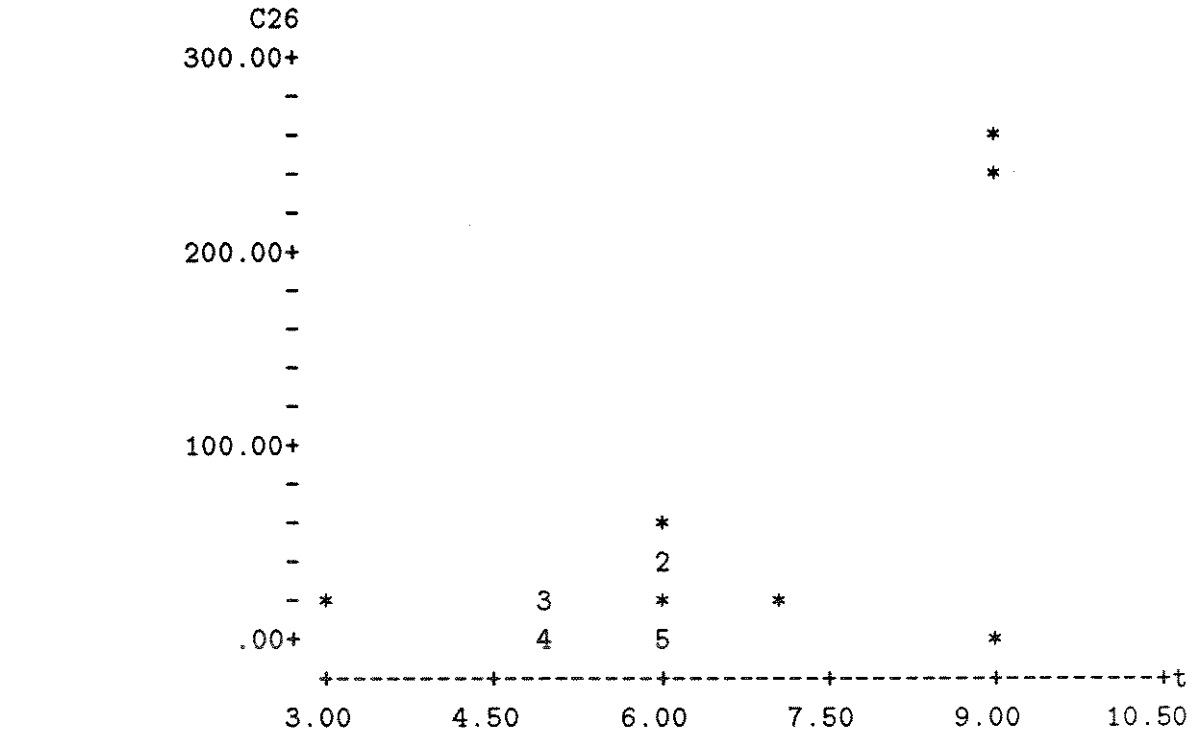



```
MTB > regress c26 2 c1 c4,c41,c42
```

```
THE REGRESSION EQUATION IS
C26 = - 153 + 23.9 t + 7.66 nodef

                                ST. DEV.     T-RATIO =
COLUMN       COEFFICIENT        OF COEF.     COEF/S.D.
                -153.33           32.34         -4.74
t                23.909           5.811          4.11
nodef             7.655           3.351          2.28

S = 46.73

R-SQUARED = 71.4 PERCENT
R-SQUARED = 68.6 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

 DUE TO      DF           SS       MS=SS/DF
REGRESSION    2       109193          54597
RESIDUAL     20        43673           2184
TOTAL        22       152866

DURBIN-WATSON STATISTIC = 2.11
MTB > plot c41 c42
```



```
MTB > omit 2 in c23, corr. c1-c4,c11,c12,c26,put c23,c1-c4,c11,c12,c26
MTB > omit 87 in c12, corr. c1-c4,c11,c26,c23,put c12,c1-c4,c11,c26,c23
MTB > choose 0 11 in c1,corr c2-c4,c11,c12,c26,c23,put c1-c4,c11,c12,c26, c23
MTB > omit 39  in c23,corr c1-c4,c11,c12,c26,put c23,c1-c4,c11,c12,c26
MTB > omit 177 in c12, corr. c1-c4,c11,c26,c23,put c12,c1-c4,c11,c26,c23
MTB > print c1 c2 c3 c4 c11 c12 c26
MTB > plot c26 c1
```



```
MTB > regress c26 1 c12,c41,c42

THE REGRESSION EQUATION IS
C26 = - 14.3 + 0.766 ducam

                                ST. DEV.    T-RATIO =
COLUMN      COEFFICIENT         OF COEF.    COEF/S.D.
               -14.321            1.679       -8.53
ducam          0.76629          0.01485       51.61

S = 6.226

R-SQUARED = 99.3 PERCENT
R-SQUARED = 99.3 PERCENT, ADJUSTED FOR D.F.

ANALYSIS OF VARIANCE

 DUE TO      DF           SS       MS=SS/DF
REGRESSION    1       103246         103246
RESIDUAL     19          737             39
TOTAL        20       103983

DURBIN-WATSON STATISTIC = 2.87
MTB > plot c41 c42
```