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

Critérios Potenciais Usos de Integração: Definição e Análise

Autor: **Plínio Roberto Souza Vilela**
Orientador: **Prof. Dr. Mario Jino**
Co-orientador **Prof. Dr. José C. Maldonado**

Tese submetida à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas para preenchimento parcial dos requisitos para obtenção do Título de Doutor em Engenharia Elétrica.

Este exemplar corresponde à redação final da tese defendida por Plínio Roberto Souza Vilela e aprovada pela Comissão Julgada em 13 / 04 / 1998.
Mario Jino
Orientador

abril de 1998

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

Vilela, Plínio Roberto Souza
V711c Critérios potenciais usos de integração: definição e análise. / Plínio Roberto Souza Vilela.--Campinas, SP: [s.n.], 1998.

Orientadores: Mario Jino, José C. Maldonado.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Engenharia de software. 2. Software - Desenvolvimento. 3. Software – Testes. I. Jino, Mario. II. Maldonado, José C. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Abstract

A Family of Data Flow based Testing Criteria for the Integration Testing of Programs, called Integration Potential Uses Criteria, is defined. This family includes basic and feasible criteria. An analysis of the properties of the criteria, including the subsumption relation and the fault detecting ability, reveals that a hierarchy, bridging the gap between data flow based criteria and path testing, is established. The hierarchy is preserved even in the presence of infeasible paths. A preliminary analysis for the specification of a tool to support the application of the Potential Uses Criteria for Integration Testing is presented. A conservative approach for testing programs with pointers, based on the approach defined on a previous work by Maldonado for testing programs with arrays, is proposed. This approach is usually more demanding, in terms of the number of required elements, than previously presented ones. Complementing Maldonado's empirical studies of the efficacy of the Potential Uses Criteria at the unit level, the efficacy of the Potential Uses Criteria is theoretically analyzed. This analysis became possible after the definition of a set of relations to compare the fault detecting ability of testing criteria presented by Frankl and Weyuker. As a result of this analysis it is shown that the Potential Uses Criteria established a hierarchy between branch testing and path testing even when the fault detecting ability of the criteria is considered.

Resumo

Uma Família de Critérios de Teste Estrutural baseado em Análise de Fluxo de Dados para o Teste de Integração de Programas, denominada Família de Critérios Potenciais Usos de Integração, é definida. Essa família de critérios inclui os critérios básicos e os critérios executáveis. As propriedades teóricas desses critérios são analisadas, incluindo a análise da relação de inclusão e da habilidade de detecção de defeitos tanto para os critérios básicos como para os executáveis. Essa família de critérios estabelece uma hierarquia de critérios entre o teste de ramos e o teste de caminhos tanto para a relação de inclusão como para a habilidade de detecção de defeitos, mesmo na presença de caminhos não executáveis. A análise inicial para a especificação de uma ferramenta de teste que suporte a aplicação dos Critérios Potenciais Usos de Integração é apresentada. Uma abordagem conservadora para o teste de programas com variáveis do tipo ponteiro, baseada na abordagem proposta por Maldonado para variáveis do tipo vetor ("array"), é apresentada. Essa abordagem é, em geral, mais exigente em termos do número de elementos requeridos que as outras abordagens apresentadas na literatura. Complementando o estudo da eficácia dos Critérios Potenciais Usos, apresentado por Maldonado, sob o ponto de vista empírico, analisa-se a eficácia desses critérios sob o ponto de vista teórico. Esse tipo de análise passou a ser possível depois da definição de um conjunto de relações para comparar a relativa habilidade de detecção de defeitos de critérios de teste apresentado por Frankl e Weyuker. Como resultado dessa análise conclui-se que os Critérios Potenciais Usos estabelecem uma hierarquia de critérios entre critérios baseados em fluxo de dados e o teste de caminhos mesmo quando a habilidade de detectar defeitos é considerada.

Agradecimentos

Eu agradeço de coração àqueles que direta ou indiretamente contribuiram para que este trabalho fosse realizado. Em especial, agradeço à minha esposa Priscila pela dedicação e pelo carinho com que suportou minhas ausências, à minha mãe Teresa, à minha irmã Flávia e ao meu pai Maury, pois a família é o nosso maior bem. Não posso deixar de agradecer à minha família por afinidade, Anita, Silmara, Sílvio e Leon.

Aos meus orientadores Mario Jino e José Maldonado os meus mais sinceros agradecimentos, sem vocês eu não teria feito nada.

Aos meus colegas do grupo de teste, Dino, Chaim, Paulo, Inês, Adalberto, Sílvia, Daniela, Claudia, Ivan, Cristina e Nelson; valeu gente!

Aos meus velhos companheiros do LCA, Nelson, Elton, Perla, Okamura, Agnus e ao meu saudoso e querido amigo Alexandre M. V. Mello, um grande abraço pra você.

Ao pessoal da Purdue University, Aditya Mathur, Cassiano, Felipe, Wang e a todos da BrLight e da Brasa.

E finalmente ao CNPq pelo apoio financeiro.

À Teresa Miyoko.

You can be anything you want to be,
just turn yourself into
anything you think that you could ever be.
Be free.

From a Queen's song.

Sumário

ABSTRACT ii

RESUMO iv

AGRADECIMENTOS vi

LISTA DE FIGURAS xvi

LISTA DE TABELAS xviii

1 Introdução 1

1.1 Contexto 1

1.2 Motivação 5

1.3 Objetivos 5

1.4 Organização da Tese 6

2 Teste Baseado em Análise de Fluxo de Dados 7

2.1 Terminologia e Conceitos Básicos 8

2.2 Critérios para o Teste de Unidade 13

2.2.1 Família de Critérios de Rapps e Weyuker 13

2.2.2 Família de Critérios Potenciais Usos 14

2.2.3 Outros Critérios de Fluxo de Dados 16

2.3 Estratégias de Teste de Software 17

2.4 Classes de Defeitos de Integração 20

2.5 Critérios para o Teste de Integração 23

2.5.1 Critérios de Linnenkugel e Müllerburg 23

2.5.2 Critérios de Harrold e Soffa 27

2.6 Comparando Critérios de Teste 33

2.7 Considerações Finais 39

3 Critérios Potenciais Usos: Análise e Extensões **41**

3.1 Teste de Programas com Ponteiros 41

3.1.1 Conceituação 42

3.1.2 Trabalhos Anteriores 43

3.1.3 Abordagem Potenciais Usos 46

3.2 Habilidade na Detecção de Defeitos 53

3.2.1 Definições 54

3.2.2 Hierarquia de Critérios 54

3.3 Considerações Finais 60

4 Extensão para o Teste de Integração **63**

4.1 Critérios Potenciais Usos para o Teste de Integração 63

4.2 Exemplos 68

4.3 Análise de Propriedades 73

4.3.1 Relação de Inclusão 74

4.3.2 Habilidade na Detecção de Defeitos 84

4.4 Considerações Finais 86

5 PokeInt - Aspectos de Implementação **89**

5.1 Reusando Informação do Teste de Unidade 89

5.1.1 Outras Informações Reusadas 91

5.2 Cálculo de Sinônimos 93

5.3 Modelos de Implementação – Integração 95

5.3.1 Modelo de Fluxo de Controle 95

5.3.2 Modelo de Instrumentação 96

5.3.3 Modelo de Fluxo de Dados 96

5.4 Arquitetura da Ferramenta PokeInt 97

5.4.1 Grafo de Chamada 97

5.4.2 Requisitos do Teste de Integração 98

5.4.3 Avaliação para o Teste de Integração 99

5.5 Considerações Finais 100

6 Conclusões **101**

6.1 Síntese do Trabalho 101

6.2 Contribuições da Tese 102

6.3 Trabalhos Futuros 103
6.3.1 Confiabilidade de Software 103
6.3.2 Teste de Programas Orientados a Objetos 104
6.3.3 Teste de Regressão no Nível de Integração 104
6.3.4 Questões Relativas ao Custo 105

Referências Bibliográficas **107**

A Teste de Unidade **115**
A.1 Modelos de Implementação para o Teste de Unidade 115
A.1.1 Modelo de Fluxo de Controle 116
A.1.2 Modelo de Instrumentação 116
A.1.3 Modelo de Fluxo de Dados 117
A.1.4 Modelo de Descrição dos Elementos Requeridos 118
A.2 Informação Gerada pela POKE-TOOL 119

Lista de Figuras

2.1 *iCAMINHOS* e *CG_CAMINHOS*. 11
2.2 Ambiente para o Teste de Unidade 18
2.3 Programa Exemplo de Harrold e Soffa. 29
2.4 Grafo de Fluxo de *GetMax* e *PairMax*. 30
2.5 Passo Inicial na Construção do Grafo de Fluxo Interprocedimental para *GetMax* e *PairMax*. 31
2.6 Grafo de Fluxo Interprocedimental para *GetMax* e *PairMax* com Acoplamento de Parâmetros Formais e Reais. 31
2.7 Grafo de Fluxo Interprocedimental para *GetMax* e *PairMax* com Conjunto de Usos Alcançáveis. 32
2.8 Relação de Inclusão. 34
2.9 Relação de Inclusão Considerando a Linguagem C. 35
2.10 Relação de Inclusão para os Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados Executáveis. 35

3.1 Programa Exemplo de Ostrand-Weyuker para Análise de Ponteiros 45
3.2 Tratamento de Ponteiros X Definição por Referência 47
3.3 Grafo de Programa – Exemplo de Ostrand e Weyuker 48
3.4 Relação Apresentada por Frankl e Weyuker 55
3.5 Exemplo – Relação Não Cobre. 57
3.6 Relação Cobre Propriamente 59

4.1 Exemplo para os Critérios Potenciais Usos de Integração. 69
4.2 Exemplo para Comparar a Abordagem de Harrold e Soffa com os Critérios Potenciais Usos de Integração. 71
4.3 Grafo de Chamada *CG* 74
4.4 Programa Exemplo # 1. 74
4.5 Programa Exemplo # 2. 77
4.6 Programa Exemplo # 3. 77

4.7 Programa Exemplo # 4. 78
4.8 Programa Exemplo # 5. 79
4.9 Programa Exemplo # 6. 80
4.10 Relação de Inclusão para Integração. 82
4.11 Relação de Inclusão dos Critérios de Integração Executáveis. 84
4.12 Divisão do Domínio de Entrada. 85
4.13 Subdomínios para Sub-caminhos e iCAMINHOS. 86
4.14 Habilidade de Detecção de Defeitos - Integração. 86

5.1 Requisitos de Teste de Integração 90
5.2 Selecionando Requisitos de Teste de Integração 90
5.3 Associações com Relevância Interprocedimental 91
5.4 Seleção de Casos de Teste para o Teste de Integração 92
5.5 Diagrama de Fluxo de Dados – Seleciona Casos de Teste. 94
5.6 Grafo de Chamada – subst.c. 95
5.7 Principais Módulos da Ferramenta PokeInt 97
5.8 Gerando o Grafo de Chamada 98
5.9 Gerando os Conjuntos de Associações 99

6.1 (A) Estrutura de Controle que Maximiza o Número de Potenciais DU-Caminhos;
(B) Exemplo para a Análise de Complexidade. 105

A.1 Informação Gerada pela POKE-TOOL 115
A.2 Modelo de Fluxo de Controle. 117
A.3 Grafo do Programa Compress() 123

Lista de Tabelas

2.1 Associações Definição-Uso Interprocedimentais entre *GetMax* e *PairMax.* . . 32
2.2 Eficácia em Revelar Defeitos e as Relações de Frankl e Weyuker. 38

3.1 Lista de Associações Requeridas 49
3.2 Associações Potenciais Usos Adicionais 50

Capítulo 1

Introdução

1.1 Contexto

Durante as três primeiras décadas da era da computação, a principal preocupação era desenvolver hardware de computador que reduzisse os custos de processamento e armazenamento de dados. O custo do hardware era predominante e a preocupação com o custo do software era secundária. A partir da década de 80, os avanços na microeletrônica resultaram em maior poder computacional e de armazenamento que acabaram por contribuir para a redução dos custos envolvidos com o hardware.

Com os computadores com um poder computacional e de armazenamento maior e com preços mais acessíveis, é natural pensar que o fator restritivo passaria a ser o software. Isso realmente aconteceu e, desde então, o software vem se tornando o componente central em muitas atividades complexas desenvolvidas em nossa sociedade. O enfoque do desenvolvimento computacional passou a ser reduzir o custo e melhorar a qualidade de soluções baseadas em computador — soluções implementadas por software. Além disso, atualmente, sistemas baseados em computação têm sido utilizados em todas as áreas da atividade humana; em conseqüência disso, aspectos de *qualidade* e *produtividade* somam-se à inerente dificuldade e à complexidade da atividade de desenvolvimento de software [Mal91].

Qualidade de software, segundo Pressman [Pre92], 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.

A Engenharia de Software tem evoluído em resposta à evolução da tecnologia de computadores e aplica métodos organizacionais e procedimentais da engenharia tradicional ao desenvolvimento de produtos de software. Pressman [Pre92] define três fases genéricas do processo de desenvolvimento de software: *definição*, *desenvolvimento* e *manutenção*. Na fase de definição, os requisitos do sistema e do software são identificados. Na fase de desenvolvimento define-se como os requisitos serão satisfeitos, o interesse principal desta tese está nessa fase; nela ocorrem três passos distintos: *projeto do software*, *codificação* e *teste do software*. A fase de manutenção concentra-se nas modificações feitas sobre o software, envolve a reaplicação das fases de definição e desenvolvimento com o objetivo de acomodar os pedidos de modificação.

Essas fases e os passos relacionados são complementados por um conjunto de outras atividades denominado *Garantia de Qualidade de Software*. Esse conjunto de atividades tem como objetivo garantir que tanto o processo de desenvolvimento como 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.

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, e constitui um dos elementos para aprimorar a produtividade e a confiabilidade do software; isto não descarta, de forma alguma, que outras atividades e procedimentos sejam conduzidos ao longo de todo o desenvolvimento de um produto; por exemplo, o uso de técnicas formais e rigorosas de especificação e de verificação. Na realidade, atividades de teste, verificação e validação têm sido introduzidas ao longo de todo o processo de desenvolvimento [Mal91].

Segundo Myers [Mye79], teste é o processo de executar um programa ou sistema com a finalidade de encontrar defeitos. Outra definição que se encontra para teste: o processo de se experimentar ou avaliar um sistema por meios manuais ou automáticos de modo a verificar se ele atende às necessidades especificadas, ou de modo a identificar as diferenças entre os resultados esperados e os reais.

Ainda, à medida que os sistemas de software cresceram em tamanho e complexidade, o esforço requerido para testar esses sistemas tem crescido exponencialmente, resultando em altos custos e em produtos com baixa confiabilidade. Tem-se verificado que embora gaste-se até 40% do orçamento para desenvolvimento do software em atividades de teste, um número significativo de defeitos permanece sem ser detectado nos produtos liberados; esses defeitos normalmente têm um impacto substancial na operação normal do sistema [Pre92].

Teste de software envolve: planejamento, projeto de casos de teste, execução e avaliação dos resultados. O planejamento da atividade de teste deve fazer parte do planejamento global do sistema, dando origem a um *Plano de Teste*, documento fundamental para o desenvolvimento do software. No Plano de Teste são estimados recursos e definidos métodos e técnicas de teste, além de uma estratégia de teste, caracterizando-se um critério de aceitação do software em desenvolvimento. O projeto de casos de teste concentra-se em um conjunto de técnicas, critérios e métodos para elaborar os casos de teste. Esses métodos, critérios e técnicas fornecem ao projetista de software uma abordagem sistemática aos testes; além disso, constituem um mecanismo que pode auxiliar a garantir a completitude dos testes e uma maior probabilidade de revelar defeitos no software [Pre92]. Em geral, deve-se projetar casos de teste que tenham a maior probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço [Mal91].

Do ponto de vista psicológico, as atividades do ciclo de vida do software anteriores ao teste (análise, projeto e codificação) são atividades construtivas. O teste, ao contrário, tem um caráter destrutivo no sentido de que procura mostrar que o produto construído possui defeitos e não atende aos requisitos para ele especificados. Isto explica por que desenvolvedores, quando empenhados na atividade de teste, procuram mostrar, através de casos de teste que não revelem a presença de defeitos, que o software por eles construído funciona da maneira especificada, em detrimento dos casos de teste que poderiam mostrar a existência de defeitos. Na verdade, casos de teste que revelam a presença de um defeito

deveriam ser o foco principal do teste. Quanto ao projeto de casos de teste, sabe-se também que é, em geral, impossível testar-se um programa para todo o seu domínio de entrada – daí dizer-se que o teste não é capaz de provar que um programa está correto.

Assim como as demais etapas do ciclo de vida do software, a atividade de teste requer planejamento e sistematização em sua execução. Uma *estratégia de teste* estabelece uma seqüência de passos a serem seguidos, nos quais técnicas específicas de teste são definidas e aplicadas. Diversas estratégias de teste têm sido propostas, fornecendo ao testador um modelo para a condução da atividade de teste. Segundo uma estratégia de teste proposta por Pressman [Pre92], a atividade de teste é dividida em quatro fases distintas: i) Teste de Unidade; ii) Teste de Integração; iii) Teste de Validação; e iv) Teste de Sistema. O Teste de Unidade visa identificar defeitos de lógica e implementação na menor unidade do projeto do software: o módulo. O Teste de Integração visa identificar defeitos na interação entre os módulos e constitui uma técnica para integrar os módulos. O Teste de Validação encerra a etapa de integração dos módulos e visa testar o programa como um todo. O Teste de Sistema visa identificar defeitos de funcionalidade e/ou desempenho do programa implantado no ambiente de operação.

Neste ponto, portanto, chegamos às seguintes dúvidas: Como os dados de teste devem ser selecionados? Como se pode dizer se um programa P foi testado suficientemente? E como comparar diferentes conjuntos de casos de teste? Define-se *método de seleção de dados de teste* como um procedimento para escolher casos de teste; e *critério de adequação dos dados de teste* como um predicado usado na avaliação do(s) dado(s) de teste, usado para se determinar o momento de parar a atividade de teste ou como mecanismo de comparação de diferentes conjuntos de casos de teste [Fra87].

Critérios de adequação de dados de teste podem ser definidos de forma a considerar apenas a especificação do programa para derivar os requisitos de teste [How87, SC96]; esses critérios são chamados *Critérios Funcionais*. Quando informações a respeito da implementação do programa são usadas, os critérios são chamados *Critérios Estruturais* [How75, WHH80, RW82, LK83, Nta84, RW85, UY88, Mal91, TLK92, PBC93, HR94, MF96]. Outra classe de critérios são os chamados *Critérios Baseados em Defeitos* [GG75, DLS78, WO80, CHR82, DMM96, Del97] que utilizam informações de defeitos típicos para derivar requisitos de teste; a Análise de Mutantes [DLS78] é um exemplo dessa classe de critérios.

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 [Mal91]. 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 conjuntamente, como um bloco, 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 tem sido aplicada, historicamente, ao teste de unidade. Os primeiros critérios de teste estrutural para o teste de unidade utilizados eram baseados unicamente no fluxo de controle de programas, sendo os mais conhecidos: o Critério Todos-Nós, o Critério Todos-Arcos e o Critério Todos-Caminhos. O Critério Todos-Nós requer que todos os comandos sejam exercitados pelo menos uma vez; o Critério Todos-Arcos requer que toda transferência de controle entre blocos de comandos seja exercitada pelo menos um vez; por sua vez o Critério Todos-Caminhos requer que todos os caminhos possíveis do programa sejam exercitados [How75, Kin76, How78, WHH80]. Existem algumas variações do Critério Todos-Arcos; uma dessas variações exige que as expressões de um comando de

decisão sejam consideradas separadamente durante o teste, é o critério Cobertura de Decisão ("Decision Coverage"). Nesta tese o Critério Todos-Arcos e suas variações serão chamados, genericamente, de *critérios de cobertura de ramos*, e o teste aplicado segundo esses critérios de *teste de ramos*.

Surgiram também os critérios baseados em análise de fluxo de dados do programa. Esses critérios usam informações de fluxo de dados do programa para derivar os requisitos de teste; em geral classificam as ocorrências de variáveis em um programa como sendo uma *definição* ou como um *uso*. Um dos precursores dos critérios baseados em Análise de Fluxo de Dados foi Herman [Her76]. Muitos outros o sucederam [LK83, Nta84, RW85, UY88, Mal91]. De acordo com Rapps e Weyuker [RW85] a motivação por trás dos critérios baseados em análise de fluxo de dados foi a indicação de que, mesmo para programas pequenos, o teste de ramos não revela a presença de defeitos simples e triviais; e que, em geral, o teste de todos os caminhos é impraticável. O objetivo principal da introdução dos critérios baseados em análise de fluxo de dados é o de fornecer uma hierarquia de critérios entre o Critério Todos-Arcos e o Critérios Todos-Caminhos e tornar o teste estrutural mais rigoroso [Mal91].

Um dos aspectos essenciais da abordagem baseada em análise de fluxo de dados é a definição e identificação de associações de fluxo de dados. A identificação de associações de fluxo de dados é complicada pela ocorrência de variáveis do tipo ponteiro e vetores, ocorrências comuns em programas reais. Alguns trabalhos foram publicados no sentido de estender ou adaptar abordagens para o teste de programas baseadas em análise de fluxo de dados para o teste de programas com uso extensivo de variáveis do tipo ponteiro e vetores [LR90, OW91, HL91, LR92, HGN93, PLR94, MF96, VMJ97b].

Os trabalhos de definição de critérios de teste estruturais baseados em análise de fluxo de dados para o teste de unidade de programas iniciaram-se na década de 70 e multiplicaram-se nas décadas de 80 e início da década de 90. Estudos comparativos entre esses critérios têm sido conduzidos a partir de meados da década de 80; apoiados, principalmente, por uma relação de inclusão e pelo estudo da complexidade dos critérios [CHR82, RW85, Wey84, Nta88, Mal91]. Mais recentemente alguns estudos foram conduzidos para estabelecer uma hierarquia entre critérios de teste segundo sua relativa habilidade de detectar defeitos [FW93b, FW93a]. Estudos empíricos têm sido conduzidos com o objetivo principal de investigar o custo de aplicação de critérios de teste baseados em análise de fluxo de dados [Wey88, BS89, Wey90, MW94a, OPTZ96, FWH97, SMV97].

O esforço de implementação de ferramentas de teste para apoiar a aplicação dos critérios baseados em análise de fluxo de dados para o teste de unidade acompanha a própria definição dos critérios de teste (alguns exemplos são: [Fra87, KL85, WS85, Cha91]), uma vez que sem o apoio de uma ferramenta automatizada a aplicação dos critérios é limitada a programas muito pequenos [KL85].

Os critérios baseados em análise de fluxo de dados passaram, portanto, por algumas fases distintas, resumidas em: definição, estudo de propriedades (comparações) e implementação (ferramentas).

A partir do início da década de 90 vem crescendo o interesse pela fase seguinte ao teste de unidade, o teste de integração. Esse interesse é refletido no surgimento de trabalhos que estabelecem técnicas para avaliação e seleção de conjuntos de casos de teste no nível interprocedimental, para serem aplicados no teste de integração [LW90, LM90, HS91, Sil94, DMM96, Del97].

O escopo principal desta tese é o de definir e analisar as propriedades teóricas de uma Família de Critérios de Teste Estrutural baseado em Análise de Fluxo de Dados para

o Teste de Integração de programas. Os critérios definidos e a análise de suas propriedades são apresentados no Capítulo 4. Os estudos iniciais para a especificação de uma ferramenta para apoiar a aplicação desses critérios são apresentados no Capítulo 5.

1.2 Motivação

Alguns pontos levantados na caracterização do contexto em que esta tese se insere podem ser destacados como determinantes para a condução deste trabalho de pesquisa.

Com o aumento da influência do custo do software sobre o custo do desenvolvimento de sistemas baseados em computador, a preocupação com aspectos de qualidade e produtividade de software vem crescendo. Além disso, produtos de software têm sido utilizados cada vez mais em aplicações essenciais para as atividades humanas. Muitas vezes atividades de alto risco dependem, quase que exclusivamente, do funcionamento confiável de produtos de software. Essa situação caracteriza a necessidade de se aplicar técnicas e ferramentas para a obtenção de produtos confiáveis e de baixo custo. Tendo em vista que defeitos estão presentes na maioria dos produtos de software produzidos e liberados, apesar dos significativos avanços da Engenharia de Software, é de fundamental importância o estudo e a caracterização de técnicas de teste de software.

O teste de software compromete até 50% do orçamento para desenvolvimento de software e mesmo assim um número significativo de defeitos permanecem sem serem detectados [Pre92]. Considerando-de que até 40% dos defeitos encontrados em um software são defeitos de integração [HS91], motiva-se o estudo e a definição de critérios de teste que apóiem a seleção de casos de teste para o teste de integração de programas.

1.3 Objetivos

O interesse principal desta tese concentra-se no estudo de técnicas estruturais de teste de software, em especial no teste estrutural baseado em análise de fluxo de dados para o teste de integração de programas.

Critérios de teste são definidos para permitir que se tenha uma abordagem mais sistemática aos testes. Os critérios de teste definem elementos no programa, constituindo requisitos que devem ser satisfeitos para que se possa considerar os testes encerrados, garantindo-se que um conjunto específico de características foi exercitado. A análise de fluxo de dados estabelece esses requisitos através da determinação das ocorrências de variáveis dentro de um programa – geralmente analisam-se os pontos onde uma variável recebeu um valor em conjunto com os pontos onde esse valor foi utilizado.

O objetivo principal é definir e analisar as propriedades teóricas de uma extensão dos Critérios Potenciais Usos para o teste de integração de programas; essa nova família de critérios recebe o nome de Critérios Potenciais Usos de Integração. Os Critérios Potenciais Usos constituem uma família de critérios estruturais baseada em análise de fluxo de dados para o teste de unidade, definida por Maldonado [Mal91].

Outro objetivo desta tese é estabelecer uma abordagem para o teste de programas com ponteiros, baseada na abordagem para variáveis do tipo vetor apresentada por Maldonado [Mal91].

Maldonado [Mal91] define e analisa as propriedades dos Critérios Potenciais Usos.

Uma das propriedades analisadas é a relação de inclusão. É demonstrado que, segundo essa relação, os Critérios Potenciais Usos mantem uma hierarquia entre o teste de ramos e o teste de caminhos, mesmo na presença de caminhos não executáveis. Além disso, nenhum outro critério de teste baseado em análise de fluxo de dados inclui os Critérios Potenciais Usos. A análise de inclusão não reflete diretamente a eficácia dos critérios comparados. No entanto, até aquele momento, não havia sido definida alguma forma de se comparar teoricamente critérios de teste segundo a sua eficácia: como conseqüencia, a eficácia dos Critérios Potenciais Usos foi analisada sob o ponto de vista empírico. Frankl e Weyuker [FW93b, FW93a] definiram uma série de relações para comparar critérios de teste segundo sua relativa habilidade de detectar defeitos; a partir de então a eficácia dos critérios de teste pode ser analisada sob o ponto de vista teórico. Nesta tese complementa-se o trabalho de análise de propriedades dos Critérios Potenciais Usos, conduzido por Maldonado, através do estudo da habilidade de detecção de defeitos desses critérios, analisada sob o ponto de vista teórico.

Características de automatização são essenciais para a aplicação prática de qualquer critério de teste. É também objetivo desta tese analisar os aspectos básicos de implementação dos Critérios Potenciais Usos de Integração definidos.

1.4 Organização da Tese

Neste capítulo foram estabelecidos o contexto, as motivações e os objetivos desta tese; caracterizou-se sua relevância dentro do contexto da engenharia de software e, em especial, para as atividades de teste de software.

No Capítulo 2 é apresentada uma revisão bibliográfica incluindo conceitos básicos sobre o teste estrutural baseado em análise de fluxo de dados e os principais critérios de teste definidos tanto para o teste de unidade quanto para o teste de integração. Esses critérios foram reescritos, segundo uma terminologia também apresentada no próprio Capítulo 2, para facilitar a comparação entre eles.

No Capítulo 3 é proposta uma abordagem para o tratamento de variáveis do tipo ponteiro dentro do escopo dos Critérios Potenciais Usos; além disso, os Critérios Potenciais Usos são analisados segundo sua habilidade de detecção de defeitos.

A definição dos Critérios Potenciais Usos de Integração é apresentada no Capítulo 4. Nesse mesmo Capítulo são também analisadas as principais características teóricas dos Critérios Potenciais Usos de Integração.

No Capítulo 5 são apresentados os principais aspectos relacionados com a implementação de uma ferramenta para automatizar a aplicação dos Critérios Potenciais Usos de Integração. Algumas informações relevantes para a implementação desses critérios, mas que dizem respeito ao teste de unidade são sumarizadas no Apêndice A.1; esse apêndice apresenta os modelos de implementação para o teste de unidade e as informações geradas pela ferramenta POKE-TOOL durante o teste de unidade. A informação apresentada no Apêndice A.1 foi extraída de [Mal91, Cha91] ou gerada automaticamente pela ferramenta POKE-TOOL e não constitui uma contribuição desta tese, mas é fundamental para a compreensão dos tópicos abordados.

Por fim, no Capítulo 6 são apresentadas as conclusões deste trabalho e perspectivas de trabalhos futuros.

Capítulo 2

Teste Baseado em Análise de Fluxo de Dados

Neste capítulo são apresentados inicialmente alguns conceitos gerais sobre Teste Estrutural baseado em Análise de Fluxo de Dados. Em seguida, é introduzida uma terminologia para ser usada na definição dos Critérios baseados em Análise de Fluxo de Dados, tanto para o teste de unidade como para o teste de integração. Alguns critérios para o teste de unidade e para o teste de integração são também apresentados neste capítulo. O uso de uma mesma terminologia para definir critérios de teste facilita o seu entendimento e a comparação entre eles. As formas de se comparar critérios de teste envolvem a análise de sua eficácia, sua força e seu custo; essas formas de comparação são apresentadas ao final do capítulo.

Retomando alguns pontos do capítulo anterior, uma das metas da Engenharia de Software é produzir software de alta qualidade. A atividade de teste desempenha um papel relevante no desenvolvimento de software. Através do teste aplicado de maneira rigorosa, tem-se a chance de identificar a presença de defeitos no software e, desta forma, contribuir decisivamente para a melhoria da sua qualidade.

Diz-se que um programa P é correto com respeito a uma especificação S se, para qualquer item de dado d pertencente ao domínio de entrada D do programa P, o comportamento do programa está de acordo com o esperado em S [Pre92]. No teste de programa, pressupõe-se que o testador ou algum mecanismo - um oráculo - possa determinar, para qualquer item de dado de entrada, se a saída do programa é a saída esperada segundo a especificação, dentro de limites de tempo e de esforço razoáveis. O oráculo, portanto, é um mecanismo pelo qual decide-se se a saída obtida corresponde à saída esperada de acordo com a especificação; o par formado pela entrada fornecida e a saída esperada, correspondente, é chamado *caso de teste*.

Sabe-se que o teste é, em geral, incapaz de provar a correção de um programa; mesmo programas simples podem ter um número muito grande de valores possíveis de entrada – *domínio de entrada* muito grande [Mye79]. Por exemplo, considere um programa que possui duas entradas inteiras x e y e uma saída z; em um computador de 32 bits existiriam $2^{32} \times 2^{32} = 2^{64}$ entradas possíveis; supondo que se leve 1 milisegundo para executar e avaliar o resultado de cada execução, levar-se-ia 58 bilhões de anos para testar o programa [Hua75]. Nesse sentido, a atividade de teste tem como principal objetivo refutar a assertiva de que o programa está correto, através de sua execução com um dado de entrada específico (um elemento do domínio de entrada) capaz de demonstrar que a saída obtida difere daquela esperada segundo a especificação (saída esperada). Os Critérios de Teste de Software concentram-se então em definir *requisitos de teste* que devem ser satisfeitos, através da elaboração de casos de teste que exercitem esses requisitos.

Os métodos utilizados para testar software são baseados, essencialmente, nas técnicas

funcional, estrutural e baseada em defeitos. A técnica funcional procura estabelecer requisitos de teste derivados da especificação funcional do software enquanto a técnica estrutural apóia-se em informações derivadas diretamente de sua implementação. A técnica baseada em defeitos, menos conhecida mas também relevante, utiliza informação de defeitos típicos de desenvolvimento para derivar requisitos de teste.

Como apresentado no capítulo anterior, os primeiros critérios de teste estrutural definidos baseavam-se exclusivamente no fluxo de controle do programa para derivar requisitos de teste. Esses critérios reforçam a idéia de que não se pode confiar em uma parte do programa que não tenha sido exercitada; fazem parte dessa linha os Critérios Todos-Caminhos, Todos-Arcos e Todos-Nós. O Critério Todos-Nós requer que todos os comandos do programa sejam exercitados pelo menos uma vez; o Critérios Todos-Arcos requer que toda transferência de controle entre blocos de comandos seja exercitada pelo menos uma vez; por fim, o Critério Todos-Caminhos requer que todos os caminhos de execução existentes em um programa sejam exercitados [How75, Kin76, How78, WHH80]. Em seguida, surgiram as técnicas baseadas em análise de fluxo de dados do programa, derivadas da idéia de que não se pode confiar em um resultado de uma computação se o ponto onde ela foi realizada e o subseqüente ponto onde o valor é usado, não foram testados conjuntamente. Com a definição de critérios baseados em análise de fluxo de dados pretende-se construir uma ponte de ligação entre o Critério Todos-Arcos e o Critério Todos-Caminhos, disponibilizando critérios mais eficazes que Todos-Arcos e menos exigentes que Todos-Caminhos.

Com a definição de um grande número de critérios, tanto os estruturais, baseados em fluxo de controle ou fluxo de dados, como os funcionais ou os baseados em defeitos, tem-se notado, nos últimos anos, um esforço para se comparar os diversos critérios de teste propostos na literatura tanto em relação a aspectos teóricos como através de experimentação.

A seguir, introduzem-se a terminologia e os conceitos básicos pertinentes ao teste estrutural de programas, em especial para a análise de fluxo de dados, tanto para o teste de unidade como para o teste de integração. Critérios de teste para o teste de unidade, relevantes para o tema principal desta tese, são apresentados na Seção 2.2. Trabalhos de pesquisa relacionados ao teste de integração são apresentados na Seção 2.5. As formas de se analisar a força, a eficácia e o custo de critérios de teste, sob o ponto de vista teórico, são apresentadas na Seção 2.6.

2.1 Terminologia e Conceitos Básicos

A terminologia apresentada nesta seção está baseada na proposta por Clarke et al. [CPRZ89]. Essa terminologia foi estendida para a análise interprocedimental apresentada na Seção 2.5 e a definição dos Critérios Potenciais Usos de Integração apresentados no Capítulo 4. Apresenta-se essa terminologia com o objetivo de uniformizar a definição de critérios de teste da literatura; esta uniformização facilita o entendimento e a comparação entre os critérios.

Uma noção importante é a diferença entre "defeito", "erro" e "falha". *Defeito* é uma deficiência mecânica ou algorítmica que se ativada pode levar a uma falha; diz-se que o defeito é inerente ao programa e é geralmente inserido por um "engano" do programador ou do analista. *Erro* é um item de informação ou estado inconsistente do programa; é um item de informação dinâmico que geralmente surge depois de um defeito ter sido exercitado. *Falha* é o evento notável onde o programa viola suas especificações; uma condição indispensável para a ocorrência de uma falha é que a execução tenha exercitado um defeito. Com essa terminologia

define-se que o objetivo do teste é revelar a presença de defeitos em um programa, ou frustrando-se esse objetivo, aumentar a confiança que se tem sobre a sua qualidade. Já a "depuração" é uma conseqüência não previsível do teste; após detectada a presença do defeito deve-se procurar identificá-lo e corrigí-lo.

Um *módulo* M pode ser tanto o programa principal como um sub-programa e, no contexto desta tese, tem apenas um *ponto de entrada* e um *ponto de saída*. Um módulo é representado por um grafo direcionado que descreve o possível fluxo de controle do módulo. Um *grafo de fluxo de controle* ou *grafo de programa* de um módulo M é um grafo direcionado G(M)=(N, E, n_{in}, n_{out}), onde N é o conjunto de nós; $E \subseteq N \times N$ é o conjunto de arcos; $n_{in} \in N$ é chamado de *nó de entrada*, e $n_{out} \in N$ é chamado de *nó de saída*. Cada nó do grafo de programa está associado a um bloco de comandos do módulo que são sempre executados conjuntamente; isto é, se um comando pertencente a um bloco é executado, todos os demais são também executados, na ordem dada. Os arcos do grafo de programa representam possíveis transferências de controle entre os nós; ou seja, se após a execução de um nó m o nó n pode ser executado haverá um arco (m, n) no grafo de programa. Na definição dos Critérios Potenciais Usos de Integração assume-se que blocos de comandos executados conjuntamente, não contendo chamadas a procedimentos, formam um único nó; cada chamada a procedimento é isolada em um nó.

O grafo de programa define os possíveis caminhos de um módulo. Um *sub-caminho* em G(M) é uma seqüência finita, possivelmente vazia, de nós $p = (n_1, n_2, \cdots, n_k)$ tal que, para todo i, $1 \leq i < k$, $(n_i, n_{i+1}) \in E$. Um sub-caminho formado pela concatenação de dois sub-caminhos p_1 e p_2 é denotado por $p_1 \cdot p_2$. Um *sub-caminho inicial* é um sub-caminho cujo primeiro nó é o nó de entrada n_{in}. Um *sub-caminho final* é um sub-caminho cujo último nó é o nó de saída n_{out}. Um *caminho* é ao mesmo tempo um sub-caminho inicial e final. O conjunto de todos os caminhos em G(M) é denotado por $CAMINHOS(M)$.

Um *ciclo* é um sub-caminho de comprimento [1] maior ou igual a dois que começa e termina no mesmo nó. Um ciclo $(n) \cdot p \cdot (n)$ de tal forma que os nós em p são distintos e não incluem n é chamado um *ciclo simples*. Um sub-caminho livre de ciclo ou ciclo simples é chamado *sub-caminho simples*.

As ocorrências de variáveis em um programa podem ser de três tipos: uma *definição*, um *uso* ou uma *indefinição*. Seja x uma variável em um módulo M. Uma *definição* de x está associada a cada nó n em G(M) que contenha um fragmento de comando que possa atribuir um valor a x; esta definição é denotada por $d_n(M, x)$. O conjunto de variáveis para as quais existe uma definição associada a um nó particular n em G(M) é denotado por $DEFINIDO(M, n)$. Um *uso* de x está associado com cada nó n ou arco (n, m) em G(M) que contenha um fragmento de comando que possa usar o valor de x. Uma indefinição de variável ocorre quando não se tem acesso ao valor da variável ou sua localização deixa de estar definida na memória.

Os usos de variáveis são classificados em dois tipos — 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. O c-uso está associado a um nó n e é representado por c-uso$_n(M, x)$. O p-uso está associado a um arco (n_1, n_2) e é representado por p-uso$_{(n_1,n_2)}(M, x)$, um p-uso de x no arco (n_1, n_2) do módulo M.

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

[1] O *comprimento* de um sub-caminho é o número de nós pertencentes ao sub-caminho

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 mesmo nó precedendo esse c-uso; caso contrário, é um *c-uso local.* Nota-se que como o p-uso está associado a um arco não é necessário se fazer a distinção entre p-uso local e p-uso global.

Um *sub-caminho livre de definição* com relação à (c.r.a) variável x é um sub-caminho p tal que para todo nó n em p, $x \notin DEFINIDO(M, n)$. Uma definição $d_m(M, x)$ *alcança* um c-uso$_n(M, x)$ se e somente se existe um sub-caminho $(m) \cdot p \cdot (n)$ tal que p é livre de definição c.r.a x e c-uso$_n(M, x)$ é um c-uso global de x. Uma definição $d_m(M, x)$ *alcança* um p-uso$_{n_1,n_2}(M, x)$ se e somente se existe um sub-caminho $(m) \cdot p \cdot (n_1, n_2)$ tal que $p \cdot (n)$ é livre de definição c.r.a x. Uma definição $d_n(M, x)$ está *viva* em um nó j se e somente se existe um sub-caminho $(n) \cdot p \cdot (j)$ onde $p \cdot (j)$ é livre de definição c.r.a x. Uma definição $d_n(M, x)$ está *viva* em um arco (j, k) se e somente se existe um sub-caminho $(n) \cdot p \cdot (j, k)$ onde $p \cdot (j)$ é livre de definição c.r.a x.

Um sub-caminho $(n_1) \cdot q \cdot (n_j, n_k)$ 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 global de x e $(n_1) \cdot q \cdot (n_j, n_k)$ é um sub-caminho simples e $q \cdot n_j$ é livre de definição c.r.a x; ou (2) (n_j, n_k) tem um p-uso de x e $q \cdot (n_j)$ é um sub-caminho livre de definição c.r.a x e $(n_1) \cdot q \cdot (n_j)$ é um sub-caminho livre de ciclo.

Uma *associação definição-c-uso* é uma tripla (i, j, x) onde i é um nó que contém uma definição global de x; j contém um c-uso global da variável x; e existe um sub-caminho livre de definição c.r.a x de i para j. Uma *associação definição-p-uso* é uma tripla $(i, (j, k), x)$ onde i é um nó que contém uma definição global de x; (j, k) contém um p-uso da variável x; e existe um sub-caminho livre de definição c.r.a x do nó i para o arco (j, k). Uma *associação de fluxo de dados* ou *associação definição-uso* é uma associação definição-c-uso, uma associação definição-p-uso ou um DU-Caminho.

Um programa é uma coleção de módulos. Um programa P é representado por um *multi-grafo direcionado* $CG(P) = (\mathcal{N}, \mathcal{E}, s)$, onde módulos são associados a nós $n \in \mathcal{N}$ e os possíveis fluxos de controle entre módulos estão associados a arcos $e \in \mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$. Como CG é um multi-grafo, pode existir mais de um arco entre dois módulos; se existir pelo menos um arco entre M_1 e M_2 existe uma *conexão* entre esses dois módulos. Assume-se que qualquer nó no grafo pode ser executado a partir do nó s, chamado *nó raiz.* O grafo CG representando um programa P é chamado de *grafo de chamada.* Cada arco $e \in \mathcal{E}$ representa um par (M_{1_i}, M_{2_i}), $1 \leq i \leq k$, uma das k chamadas ao módulo M_2 no módulo M_1. O módulo M_1 é o *módulo chamador* e M_2 é o *módulo chamado.* O conjunto de todos os caminhos entre dois módulos M_1 e M_2 de $CG(P)$, considerando a concatenação de $G(M_1)$ e $G(M_2)$, é denotado por $iCAMINHOS(M_1, M_2)$. O conjunto de todos os caminhos no grafo de chamada $CG(P)$ é denotado por $CG\_CAMINHOS(P)$.

Observa-se na Figura 2.1 a diferença entre $iCAMINHOS$ e $CG\_CAMINHOS$. Para os módulos A e B da figura, o conjunto $CG\_CAMINHOS$ é composto por um único caminho entre o módulo A e o módulo B. Por sua vez, o conjunto $iCAMINHOS(A, B)$ contém os seguintes caminhos:

- $n_{in}(A)\ 1_a 2_a 4_a\ n_{in}(B)\ 1_b 2_b 4_b\ n_{out}(B)\ 5_a 6_a 8_a\ n_{out}(A)$
- $n_{in}(A)\ 1_a 2_a 4_a\ n_{in}(B)\ 1_b 2_b 4_b\ n_{out}(B)\ 5_a 7_a 8_a\ n_{out}(A)$
- $n_{in}(A)\ 1_a 2_a 4_a\ n_{in}(B)\ 1_b 3_b 4_b\ n_{out}(B)\ 5_a 6_a 8_a\ n_{out}(A)$
- $n_{in}(A)\ 1_a 2_a 4_a\ n_{in}(B)\ 1_b 3_b 4_b\ n_{out}(B)\ 5_a 7_a 8_a\ n_{out}(A)$
- $n_{in}(A)\ 1_a 3_a 4_a\ n_{in}(B)\ 1_b 2_b 4_b\ n_{out}(B)\ 5_a 6_a 8_a\ n_{out}(A)$

- $n_{in}(A)\ 1_a 3_a 4_a\ n_{in}(B)\ 1_b 2_b 4_b\ n_{out}(B)\ 5_a 7_a 8_a\ n_{out}(A)$
- $n_{in}(A)\ 1_a 3_a 4_a\ n_{in}(B)\ 1_b 3_b 4_b\ n_{out}(B)\ 5_a 6_a 8_a\ n_{out}(A)$
- $n_{in}(A)\ 1_a 3_a 4_a\ n_{in}(B)\ 1_b 3_b 4_b\ n_{out}(B)\ 5_a 7_a 8_a\ n_{out}(A)$

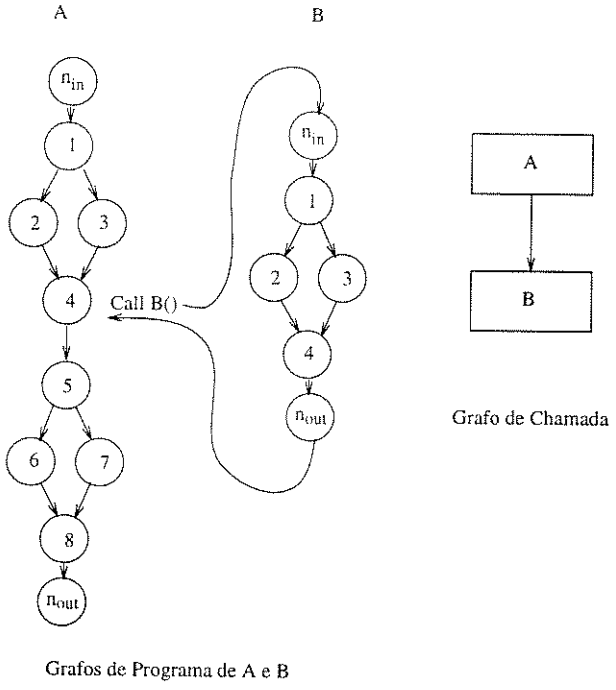

Figura 2.1: *iCAMINHOS* e *CG_CAMINHOS*.

O parâmetro formal correspondente a um parâmetro real x de uma chamada c é denotado por $formal_c(x)$. Para simplificar as definições considere que no caso de uma variável global x, $formal_c(x)$ mapeia a variável para ela mesma; $formal_c(x) = x$.

O nó em um módulo M_1 onde a chamada c ocorre é denotado por n_c. O conjunto de variáveis usados como entrada ou saída de uma chamada c do módulo M_1 ao módulo M_2 é denotado por in_c que é o conjunto de parâmetros reais ou variáveis globais usados como entrada na chamada c do módulo M_1 ao módulo M_2; out_c é o conjunto de parâmetros reais ou variáveis globais usados como saída na chamada c do módulo M_1 ao módulo M_2. O conjunto de todas as variáveis $in_c \cup out_c$ é genericamente chamado de *variáveis de comunicação.*

Uma *associação-definição-c-uso interprocedimental de chamada* é uma quádrupla (i, j, x, c) onde c é uma chamada de um módulo M_1 para um módulo M_2; $x \in in_c$; i é um nó de M_1 que contém uma definição global de x; j é um nó em M_2 que contém um c-uso global de $formal_c(x)$, e existe um sub-caminho livre de definição c.r.a x de i para n_c e c.r.a $formal_c(x)$ de $n_{in}(M_2)$ para j.

Uma *associação-definição-c-uso interprocedimental de retorno* é uma quádrupla $(i, j,$ $x, c)$ onde c é uma chamada de um módulo M_1 para um módulo M_2; $x \in out_c$; i é um nó de M_2 que contém uma definição global de $formal_c(x)$; j é um nó em M_1 que contém um c-uso global de x, e existe um sub-caminho livre de definição c.r.a $formal_c(x)$ de i para out_c e c.r.a x de n_c para j.

Uma *associação-definição-p-uso interprocedimental de chamada* é uma quádrupla $(i, (j, k), x, c)$ onde c é uma chamada de um módulo M_1 para um módulo M_2; $x \in in_c$; i é um nó de M_1 que contém uma definição global de x; (j, k) é um arco em M_2 que contém um

p-uso de $formal_c(x)$, e existe um sub-caminho livre de definição c.r.a x de i para n_c e c.r.a $formal_c(x)$ de $n_{in}(M_2)$ para (j,k).

Uma *associação-definição-p-uso interprocedimental de retorno* é uma quádrupla $(i, (j,k), x, c)$ onde c é uma chamada de um módulo M_1 para um módulo M_2; $x \in out_c$; i é um nó de M_2 que contém uma definição global de $formal_c(x)$; (j,k) é um arco em M_1 que contém um p-uso de x, e existe um sub-caminho livre de definição c.r.a $formal_c(x)$ de i para out_c e c.r.a x de n_c para (j,k).

Um sub-caminho $(n_i) \cdot q \cdot (n_c) \cdot (n_{in}(M_2)) \cdot p \cdot (n_j, n_k)$; onde n_c é o nó da chamada c de M_1 para M_2, $n_i \in M_1$ e $(n_j, n_k) \in M_2$, é um *DU-Caminho Interprocedimental de Chamada* c.r.a x, se e somente se $(n_i) \cdot q \cdot (n_c)$ é um DU-Caminho c.r.a x e $(n_{in}(M_2)) \cdot p \cdot (n_j, n_k)$ é um DU-Caminho c.r.a $formal_c(x)$.

Um sub-caminho $(n_i) \cdot q \cdot (n_{out}(M_2)) \cdot (n_c) \cdot p \cdot (n_j, n_k)$; onde n_c é o nó da chamada c de M_1 para M_2, $n_i \in M_2$ e $(n_j, n_k) \in M_1$, é um *DU-Caminho Interprocedimental de Retorno* c.r.a $formal_c(x)$, se e somente se $(n_i) \cdot q \cdot (n_{out}(M_2))$ é um DU-Caminho c.r.a $formal_c(x)$ e $(n_c) \cdot p \cdot (n_j, n_k)$ é um DU-Caminho c.r.a x.

Uma *associação de fluxo de dados interprocedimental* ou uma *associação definição-uso interprocedimental* é uma associação definição-c-uso interprocedimental de chamada ou de retorno, ou uma associação definição-p-uso interprocedimental de chamada ou de retorno, ou um DU-Caminho interprocedimental de chamada ou de retorno.

Um *critério C de seleção de caminhos para o teste de unidade*, ou simplesmente um *critério de teste de unidade*, é um predicado que atribui um valor verdadeiro ou falso para um par (M, Π), onde M é um módulo e Π é um subconjunto de $CAMINHOS(M)$. Um par (M, Π) *satisfaz* um critério de teste de unidade C se e somente se $C(M, \Pi) = verdadeiro$. Um *critério de seleção de caminhos para o teste de integração*, ou simplesmente um *critério de teste de integração*, é um predicado que atribui um valor verdadeiro a qualquer par (P, Ψ), onde P é o programa e Ψ é um subconjunto de $CG\_CAMINHOS(P)$. Se o critério de integração for *dois-a-dois* ("pairwise criterion"), tem-se que um *critério de teste de integração dois-a-dois* é um predicado que associa um valor verdadeiro a qualquer tripla (M_1, M_2, Φ), onde M_1 e M_2 são módulos e (M_1, M_2) é um multi-arco em $CG(P)$ e Φ é um subconjunto de $iCAMINHOS(M_1, M_2)$. A tripla (M_1, M_2, Φ) satisfaz um critério de teste de integração dois-a-dois C se, e somente se, $C(M_1, M_2, \Phi) = verdadeiro$. De forma equivalente pode-se dizer que um conjunto de casos de teste T satisfaz um critério C para um programa P (ou módulo M) se T implica na execução de todos os elementos requeridos estabelecidos por C em P (ou M); nesse caso diz-se que T é *C-adequado* para P (para M).

Um critério de teste C_1 *inclui* [2] um critério C_2 se e somente se qualquer par (M, Π) ou (P, Ψ) ou tripla (M_1, M_2, Φ) que satisfaz C_1 também satisfaz C_2. Dois critérios são *equivalentes* se cada um inclui o outro. Um critério C_1 *inclui estritamente* um critério C_2 se e somente se C_1 inclui C_2 mas C_2 não inclui C_1. Dois critérios são incomparáveis se nenhum inclui o outro.

Sabe-se que o fato de um critério C_1 incluir um critério C_2 não garante que C_1 seja melhor que C_2 quando a habilidade de detecção de defeitos é considerada. Frankl e Weyuker [FW93b] provaram que para a *relação cobre propriamente* ("properly covers relation"), pode-se garantir que se C_1 *cobre propriamente* C_2, C_1 é melhor em detectar defeitos que C_2, de acordo com algumas medidas probabilísticas. O foco está em como cada critério subdivide o domínio de entrada em subdomínios.

[2] A relação de inclusão foi definida por Rapps e Weyuker [RW85] para critérios de teste de unidade, na definição apresentada nesta tese o mesmo conceito é estendido para critérios de integração.

2.2 Critérios para o Teste de Unidade

Nesta seção consideram-se a Família de Critérios Baseada em Fluxo de Dados proposta por Rapps e Weyuker [RW85] e os Critérios Potenciais Usos propostos por Maldonado [Mal91] para o teste de unidade. A definição desses critérios é reescrita segundo a terminologia apresentada na seção anterior, e os critérios são comparados em termos da relação de inclusão e de sua relativa habilidade de detecção de defeitos.

Os critérios baseados em análise de fluxo de dados utilizam a informação de fluxo de dados para derivar os requisitos de teste e requerem que as interações que envolvem a definição de variáveis de programa, e subseqüentes referências a essas definições, sejam testadas. O propósito de se definir esses critérios é o de preencher a lacuna entre o teste de ramos (Critério Todos-Arcos) e o teste de caminhos (Critério Todos-Caminhos), estabelecendo critérios mais exigentes que Todos-Arcos mas que não levem à seleção de um número tão grande de elementos requeridos como o Critério Todos-Caminhos [Mal91].

2.2.1 Família de Critérios de Rapps e Weyuker

A Família de Critérios Baseada em Fluxo de Dados de Rapps e Weyuker inclui três critérios baseados em análise de fluxo de controle e alguns critérios baseados em análise de fluxo de dados. Os critérios baseados em análise de fluxo de controle são: *Todos-Caminhos*, *Todos-Arcos* e *Todos-Nós*.

> O par (M, Π) satisfaz o *Critério Todos-Caminhos* se e somente se $\Pi = CAMINHOS(M)$.
>
> O par (M, Π) satisfaz o *Critério Todos-Arcos*, se e somente se, para todo arco $e \in E$, onde E é o conjunto de arcos de $G(M)$, existe pelo menos um caminho em Π no qual e ocorre.
>
> O par (M, Π) satisfaz o *Critério Todos-Nós*, se e somente se, para todo nó $n \in N$, onde N é o conjunto de nós de $G(M)$, existe pelo menos um caminho em Π no qual n ocorre.

A idéia por trás da definição desse conjunto de critérios é a de que não se pode confiar em uma parte do código (do programa) que nunca tenha sido executada. O problema com esses critérios é que se sabe que mesmo módulos pequenos podem ter um número infinito de caminhos; desta forma, o único par (M, Π) que iria satisfazer o critério *Todos-Caminhos* seria aquele com um conjunto Π infinito; além disso, também se sabe que importantes combinações de arcos e nós não são requeridas nem pelo Critério Todos-Arcos nem pelo Critério Todos-Nós. Rapps e Weyuker então propuseram um conjunto de critérios com o objetivo de preencher a lacuna entre o critério Todos-Arcos e o critério Todos-Caminhos; esses critérios estão baseados na utilização das informações de fluxo de dados de um programa para se derivar os requisitos de teste.

O primeiro critério definido requer que cada definição de variável seja exercitada. O *Critério Todas-Definições* requer que o conjunto de caminhos contenha pelo menos um sub-caminho livre de definição a partir de cada definição até algum uso dessa definição.

> O par (M, Π) satisfaz o *Critério Todas-Definições*, se e somente se, para toda definição $d_m(M, x)$ global existir pelo menos um sub-caminho $(m) \cdot p \cdot (n)$ em Π

tal que existe um c-uso$_n(M,x)$ global e p é livre de definição c.r.a x, ou existir pelo menos um sub-caminho $(m)\cdot p\cdot(n,n')$ tal que existe um p-uso$_{(n,n')}(M,x)$ e $p\cdot(n)$ é livre de definição c.r.a x.

A idéia por trás da definição desse critério e do resto da Família de Critérios Baseados em Análise de Fluxo de Dados é a de que não se pode confiar em um resultado de uma computação que nunca foi usado. O próximo passo é requerer que todos os usos de uma dada definição sejam cobertos pelo conjunto de caminhos.

O par (M,Π) satisfaz o critério *Todos-Usos*, se e somente se, para toda definição $d_m(M,x)$ global, Π incluir pelo menos um sub-caminho $(m)\cdot p\cdot(n)$ para todo c-uso$_n(M,x)$ global com p livre de definição c.r.a x, e pelo menos um sub-caminho $(m)\cdot p\cdot(n,n')$ para todo p-uso$_{(n,n')}(M,x)$, com $p\cdot(n)$ livre de definição c.r.a x.

O critério *Todos-Usos* requer todas as associações entre um ponto do programa onde a variável recebeu um valor e todo ponto onde esse valor é usado. Rapps e Weyuker definem três critérios similares a *Todos-Usos* (*Todos-C-Usos/Alguns-P-Usos, Todos-P-Usos/Alguns-C-Usos* e *Todos-P-Usos*) mas que distinguem entre o *uso computacional* e o *uso predicativo.*

O último critério, *Todos-DU-Caminhos*, é mais "forte" que *Todos-Usos* pois, ao invés de requerer um sub-caminho livre de definição de toda definição para todo uso, ele requer todos DU-Caminhos entre a definição e todos os usos da variável.

O par (M,Π) satisfaz o *Critério Todos-DU-Caminhos*, se e somente se, para toda definição $d_m(M,x)$ global, Π incluir todo *DU-Caminho* de m para todo c-uso$_n(M,x)$ e para todo p-uso$_{(n,n')}(M,x)$.

2.2.2 Família de Critérios Potenciais Usos

Rapps e Weyuker definiram que uma associação definição-uso existe entre uma definição e um uso de uma variável. Maldonado [Mal91] modificou esse conceito e definiu que uma *potencial associação definição-uso* (Definição 2.1) existe entre uma definição e todo ponto onde essa definição está viva – existe um *potencial uso* da variável, em relação à definição, naquele ponto. A identificação de associações de fluxo de dados é reduzida à identificação de definições e usos de variáveis na abordagem de Rapps e Weyuker e à identificação de definições de variáveis na abordagem de Maldonado. Pode-se dizer que a abordagem Potencial Uso alivia uma das fraquezas do teste estrutural; não se está apenas tentando identificar defeitos devido ao uso incorreto de variáveis, mas inclui a possibilidade de se identificar defeitos devido à ausência do uso, que também constitui um defeito.

Definição 2.1 *Uma* Potencial Associação Definição Uso (n_1,n_2,v) *de um módulo* M *existe se* n_1 *tem uma definição global de* v *e existe um sub-caminho* $(n_1)\cdot p\cdot(n_2)$ *tal que* $p\cdot(n_2)$ *é livre de definição c.r.a* v; n_1 *e* n_2 *são nós do grafo de programa de* M.

Definição 2.2 *Um sub-caminho* $(n_1)\cdot q\cdot(n_j,n_k)$ *é um* Potencial DU-Caminho *c.r.a variável* x *se* n_1 *tiver uma definição global de* x *e: (1) ou* n_k *tem um potencial uso de* x *e* $(n_1)\cdot q\cdot(n_j,n_k)$ *é um sub-caminho simples e* $q\cdot(n_j)$ *é livre de definição c.r.a* x; *ou (2)* (n_j,n_k) *tem um potencial uso de* x *e* $q\cdot(n_j)$ *é um sub-caminho livre de definição c.r.a* x *e* $(n_1)\cdot q\cdot(n_j)$ *é um sub-caminho livre de ciclo.*

Essa abordagem ao teste de programas baseado em análise de fluxo de dados levou à definição de uma família de critérios (*Os Critérios Potenciais Usos*) que inclui os critérios *básicos*, os *executáveis* e os *estendidos a ciclo* [Mal91]. Os critérios básicos são definidos como se segue.

> O par (M, Π) satisfaz o critério *Todos-Potenciais-Usos*, se e somente se, para toda definição $d_m(M, x)$ global, Π incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n)$ para todo nó n com um potencial uso de x definido em $d_m(M, x)$, com p livre de definição c.r.a x, e pelo menos um sub-caminho $(m) \cdot p \cdot (n, n')$ para todo arco (n, n') com um potencial uso de x definido em $d_m(M, x)$, com $p \cdot (n)$ livre de definição c.r.a x.

A definição do critério *Todos-Potenciais-Usos* implica que todas as associações definição-uso requeridas pelo critério *Todos-Usos* também serão requeridas pelo critério *Todos-Potenciais-Usos*; o fato de que toda associação de fluxo de dados também é uma potencial associação de fluxo de dados é o que está por trás dessa propriedade.

> O par (M, Π) satisfaz o critério *Todos-Potenciais-Usos/DU*, se e somente se, para toda definição $d_m(M, x)$ global, Π incluir pelo menos um Potencial DU-Caminho para todo nó e para todo arco com um potencial uso de x alcançável a partir de $d_m(M, x)$.

O critério *Todos-Potenciais-Usos/DU* é uma "evolução" do critério *Todos-Potenciais-Usos*; os elementos requeridos continuam sendo potenciais associações de fluxo de dados, mas é colocada uma restrição para forçar um número finito de sub-caminhos que podem ser selecionados; a potencial associação deve ser exercitada por um DU-Caminho.

> O par (M, Π) satisfaz o critério *Todos-Potenciais-DU-Caminhos*, se e somente se, para toda definição $d_m(M, x)$ global, Π incluir todo Potencial DU-Caminho de m para todo nó e para todo arco com um potencial uso de x alcançável a partir de $d_m(M, x)$.

Uma propriedade interessante envolvendo a definição dos Critérios Potenciais Usos, usada para provar algumas das relações de inclusão entre esses critérios e os critérios de Rapps e Weyuker, é que, tanto no nível de unidade como no nível de integração, todo *uso* de variável é também um *potencial uso* da variável; ou seja, toda associação de fluxo de dados é também uma potencial associação de fluxo de dados.

Uma das propriedades que devem ser satisfeitas por um bom critério de teste é a *aplicabilidade* [FW86]; 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 requerido pelo critério C. Um caminho é *não executável* se não existe um elemento do domínio de entrada do programa que faça com que a execução percorra esse caminho.

Os Critérios Potenciais Usos, assim como os critérios da Família de Critérios de Fluxo de Dados, foram modificados com o intuito de satisfazer a propriedade de aplicabilidade. De acordo com a definição original desses critérios, eles não satisfazem a propriedade de aplicabilidade, já que podem exigir elementos não executáveis, não havendo, dessa forma,

nenhum conjunto T que seja *C-adequado* ao critério. A modificação introduzida nos Critérios Potenciais Usos consiste, essencialmente, em selecionar as potenciais-associações executáveis. Os critérios assim definidos são denominados Critérios Potenciais Usos Executáveis [Mal91].

Os Critérios Potenciais Usos Estendidos a Ciclo estão baseados na seguinte definição [FW86]. Seja $\Pi = (n_1, n_2, \cdots, n_k)$ um DU-Caminho (Potencial DU-Caminho) c.r.a x; a extensão-a-ciclo(Π, x) é o conjunto de caminhos livre de definição c.r.a x do seguinte tipo $(\lambda_1, \lambda_2, \cdots, \lambda_k)$ onde 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). Segue a definição do Critério Todos-Potenciais-DU-Caminhos Estendido a Ciclo ("Cycle-Extended-All-Potential-DU-Paths").

> O par (M, Π) satisfaz o critério *Todos-Potenciais-DU-Caminhos Estendido a Ciclo*, se e somente se, para toda definição $d_m(M, x)$ global, Π incluir todo Potencial DU-Caminho π de m para todo nó e para todo arco com um potencial uso de x alcançável a partir de $d_m(M, x)$ e para cada Potencial DU-Caminho π c.r.a x a partir de m, algum caminho $\pi_1 \in$ extensão-a-ciclo(π, x).

2.2.3 Outros Critérios de Fluxo de Dados

Vários critérios baseados no fluxo de controle têm sido propostos com o objetivo de serem mais rigorosos que o Critério Todos-Arcos, mas com custo de aplicação, em geral, menor que o Critério Todos-Caminhos. Uma abordagem é restringir o número de vezes que um laço pode ser executado para tornar finito o número total de caminhos possíveis [Kin76]. Woodward et al. [WHH80] motivados pela indicação de estudos empíricos de que a presença de um número significativo de defeitos pode ser revelada pela simples execução de concatenações de arcos, propôs uma hierarquia de critérios que consiste essencialmente em concatenar seqüências de código encerradas por uma transferência de controle ("LCAJ - Linear Code Sequence and Jump").

Seguindo a idéia básica de se analisar o fluxo de dados em um programa para derivar requisitos de teste, uma quantidade grande de variações foram definidas, dando origem a várias famílias de critérios. Duas dessas famílias foram analisadas mais detalhadamente neste capítulo, os Critérios de Rapps e Weyuker e os Critérios Potenciais Usos. Nesta seção são apresentados outras variações sobre a mesma idéia.

O *Critério de Herman* [Her76] foi um dos primeiros critérios a considerar informações de fluxo de dados para estabelecer requisitos de teste. O critério requer que toda referência (uso) de uma variável seja exercitada, pelo menos uma vez, a partir de pontos do programa onde essa variável é definida. Como já foi visto neste capítulo o Critério Todos-Usos de Rapps e Weyuker nada mais é que uma variação do Critério de Herman. Os outros critérios definidos por Rapps e Weyuker se utilizam da distinção entre os tipos de usos na sua definição; os usos são classificados como computacionais ou predicativos dando origem a uma série de variações de critérios. O novo conceito introduzido por Rapps e Weyuker é o de testar todos os caminhos entre um ponto de definição e o de uso de uma variável; isto é conseguido através da definição do conceito de DU-Caminho que garante um número finito de elementos requeridos.

Uma variação na análise de fluxo de dados decorre da introdução do conceito de *cadeias de associações*. Esse conceito foi introduzido por Ntafos [Nta84] e está baseado na seguinte observação: muitas vezes o uso de uma variável se dá em um comando de atribuição onde uma segunda variável está sendo definida; portanto, uma associação pode

estar "conectada" a outra subseqüente que, por sua vez, pode estar conectada a outra e, assim, sucessivamente, dando a idéia de uma cadeia de associações conectadas. Ntafos introduziu então 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 interactions") sejam testadas. Variando-se K, obtém-se a família de critérios K-tuplas requeridas.

Outras variações existem, como é o caso dos critérios definidos por Laski e Korel [LK83] baseados no conceito de *contexto elementar de dados*; a introdução desses critérios deve-se à intuição de que valores atribuídos a uma variável em um ponto particular i do programa podem depender do valor de várias variáveis, onde cada uma tem várias definições distintas que atinjam i por caminhos livres de definição.

Ural e Yang [UY88] introduziram um critério denominado *Todos Caminhos-ES Simples* ("All-Simple OI-Paths"). Este critério procura explorar efeitos entre entradas e saídas do programa, através do estabelecimento de seqüências de associações que levem à execução de um caminho (os autores denominam "caminho completo").

2.3 Estratégias de Teste de Software

Assim como as demais etapas do ciclo de vida do software, a atividade de teste requer planejamento e sistematização em sua execução. Uma *estratégia de teste* estabelece uma seqüência de passos a serem seguidos, nos quais técnicas específicas de teste são definidas e aplicadas. Diversas estratégias de teste têm sido propostas, fornecendo ao testador um modelo para condução da atividade de teste. Como regra geral, essas estratégias possuem os seguintes pontos em comum [Pre92]:

- o teste inicia no nível de módulos ou unidades, evoluindo até alcançar o teste do sistema como um todo;
- diferentes técnicas de teste são indicadas para as diferentes etapas do teste;
- o teste é conduzido pelos desenvolvedores e por uma equipe independente de teste; e
- embora o teste e a depuração sejam atividades diversas, a estratégia de teste deve acomodar a atividade de depuração.

Segundo uma estratégia de teste proposta por Pressman [Pre92], a atividade de teste é dividida em quatro fases distintas: i) Teste de Unidade; ii) Teste de Integração; iii) Teste de Validação; e iv) Teste de Sistema.

O teste de unidade concentra-se na validação da menor parte do software, ou seja, seus módulos ou unidades. Utilizando a especificação de cada módulo como guia, procura-se revelar a existência de defeitos no escopo de cada módulo isoladamente. Nessa fase, o testador deve estar preocupado com aspectos computacionais e algorítmicos da unidade, procurando garantir que toda funcionalidade dela requerida tenha sido implementada e que toda funcionalidade implementada tenha sido exercitada. Cada unidade tomada separadamente não é um programa completo. Por isso, para testar uma unidade é preciso que se construa, a partir dela, um programa que possa ser executado e validado. Essa construção se dá através da criação de um "driver" e de "stubs" de teste. O ambiente necessário para se conduzir o teste de unidade é mostrado na Figura 2.2. O *driver* é o módulo responsável por coordenar o teste. Ele deve receber os dados de teste do testador, passá-los como parâmetros para o

módulo em teste e apresentar os resultados produzidos, para que o testador possa avaliá-los. Os *stubs* servem para substituir módulos subordinados ao (ou chamados pelo) módulo em teste. Esses stubs são, em geral, módulos que simulam o comportamento de módulos reais através de um mínimo de computação ou manipulação de dados.

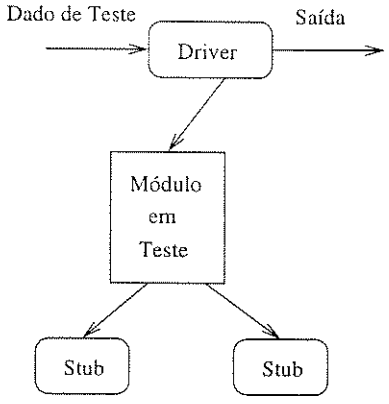

Figura 2.2: Ambiente para o Teste de Unidade

O *teste de integração* pode ser visto como uma técnica sistemática para a construção da estrutura do software, procurando revelar defeitos associados à interação entre os módulos. Mesmo tendo sido testados individualmente, quando os módulos são colocados para interagir uns com os outros, surgem problemas que são conhecidos como defeitos de integração. A fonte desses problemas está na interação entre os módulos.

A integração dos módulos pode ser feita de maneira não incremental, numa abordagem conhecida como "Integração Big-Bang". Nesse caso, todos os módulos são colocados para interagir juntos, de uma só vez, e o programa é testado como um todo. Em geral, o resultado é uma situação caótica, onde um número alto de problemas é identificado fazendo com que a correção seja difícil. Não se consegue isolar cada problema, identificar sua origem e eliminá-la.

Muitos autores recomendam que se utilizem abordagens incrementais para o teste de integração. Em especial, Pressman [Pre92] aponta que deve-se preferir as estratégias incrementais pois, segundo ele, o programa é construído e testado em pequenos segmentos, onde defeitos são mais facilmente isolados e corrigidos, as interfaces são exercitadas de maneira mais completa e uma abordagem sistemática é aplicada.

Uma das maneiras de se integrar o software é chamada de "Integração Top-Down". A integração inicia-se a partir do módulo principal de controle do programa em direção aos módulos inferiores na hierarquia de controle. A integração Top-Down tem a vantagem de que os principais pontos de decisão na hierarquia de controle são testados antes. Assim, se existem problemas sérios de controle, eles são logo reconhecidos e podem ser prontamente corrigidos. Além disso, é possível integrar-se primeiro um ramo da estrutura do software, permitindo que se possa visualizar e demonstrar algumas funcionalidades do software precocemente.

Existem também problemas na utilização dessa abordagem. O mais comum deles ocorre quando, para se testar a integração dos módulos de níveis mais altos de maneira eficaz, são necessários dados processados nos níveis mais baixos. Como stubs são usados no início da integração Top-Down, esses dados podem não estar disponíveis nesta fase da integração. Nesse caso, o testador pode decidir adiar os testes até que os stubs sejam substituídos pelos módulos reais nos níveis inferiores ou pode optar por desenvolver stubs mais completos e

que simulem a funcionalidade dos módulos reais. A primeira solução faz com que se perca o controle da relação que existe entre introduzir-se um novo módulo na estrutura e criarem-se novos casos de teste para aquele módulo. A segunda opção cria uma sobrecarga de programação para que stubs sejam desenvolvidos para o teste de integração.

Uma terceira estratégia de integração consiste em iniciar-se a integração a partir dos módulos da base da estrutura do software em direção ao seu topo. É a chamada "Integração Bottom-Up". Nesse caso, como os módulos são integrados de baixo para cima, ao integrar-se um módulo num determinado nível, sempre dispõem-se dos módulos a ele subordinados, eliminando assim a necessidade da elaboração de stubs. Dois tipos de integração Bottom-Up podem ser utilizados, a integração por grupos de módulos ("clusters") e a integração dois-a-dois ("pairwise"). Para o caso da integração dois-a-dois, cada par de módulos é integrado de cada vez, a partir dos níveis mais baixos da hierarquia do programa. Para o caso da integração por grupos de módulos, os passos da integração Bottom-Up são:

1. os módulos dos níveis mais baixos são combinados de uma vez em grupos ("clusters") que realizam uma determinada subfunção;
2. drivers são preparados para coordenar o teste dos clusters;
3. o cluster é testado; e
4. os drivers são retirados e os clusters são combinados através da integração de um módulo do nível superior.

À medida que se sobe no nível da hierarquia de controle, a necessidade de drivers diminui, até atingir-se o nível mais alto, quando o módulo principal é integrado. A principal desvantagem da integração Bottom-Up é que não se tem uma visão geral do software até que o último módulo tenha sido integrado; ou seja, até que o módulo principal de controle tenha sido incluído na estrutura e que o software esteja completamente construído [Pre92].

Muitas vezes procura-se minimizar as desvantagens de cada abordagem de integração através da combinação delas numa única estratégia. É a chamada "Integração Sanduíche". Nesse caso, utiliza-se a integração Top-Down para os módulos de níveis mais altos e a integração Bottom-Up para os níveis mais baixos, até atingir-se um ponto no meio da estrutura do software onde essas estratégias "se encontram".

Finalizado o teste de integração, o software está completamente montado, e uma série final de testes, chamados de *teste de validação*, deve ser realizada. O objetivo principal é assegurar que todos os requisitos funcionais, de performance e outros (como transportabilidade, compatibilidade, recuperação a defeitos, manutenibilidade), presentes na especificação do software, estão presentes no produto desenvolvido. A documentação de usuário deve também ser avaliada nesta etapa de teste.

Todo sistema é parte de um sistema mais abrangente. O software é apenas uma parte num sistema maior, interagindo com outros elementos desse sistema, como por exemplo, o hardware. Por isso, uma série de testes, chamados *teste de sistema*, devem ser realizados, buscando validar a integração do software com os demais elementos do sistema em que ele está inserido. Esses testes não são conduzidos unicamente pelo desenvolvedor; porém, passos tomados durante o projeto e teste do software podem facilitar sua integração em sistemas maiores.

2.4 Classes de Defeitos de Integração

O teste de integração tem o objetivo principal detectar a presença de defeitos de integração no software; esses defeitos encontram-se nas interfaces dos módulos e em suas interações, e podem constituir em até 40% dos defeitos de um programa [BP94, HS91].

Apresenta-se a seguir uma taxonomia para esse tipo de defeito [3], definida originalmente por Beizer [Bei90].

- **6xxx: INTEGRAÇÃO**: defeitos relacionados com a integração e a interface dos componentes. Os componentes propriamente ditos são assumidos como corretos.
  - **61xx: INTERFACES INTERNAS**: defeitos relacionados à interface entre componentes que se comunicam no programa em teste. Assume-se que os componentes passaram pelo teste de unidade. Neste contexto, a transferência direta ou indireta de informações de fluxo de controle ou dados por objetos de memória do tipo tabelas, recursos alocados dinamicamente, ou arquivos, constituem a interface interna.
    - **611x: Invocação de Componente**: defeitos relacionados a como os componentes do software são invocados. Neste sentido, um "componente" pode ser uma subrotina, uma função, uma macro, um programa, um segmento de programa, ou qualquer outro componente de software identificável.
      - **6111: Componente inexistente**: o componente invocado não existe.
      - **6112: Componente incorreto**: o componente invocado não é o correto.
    - **612x: Invocação de Parâmetro**: relacionado com os parâmetros do componente, seu número, ordem, tipo, localização, valores, etc.
      - **6121: Parâmetro incorreto**: os parâmetros da invocação não estão especificados corretamente.
      - **6122: Tipo do parâmetro**: tipo de parâmetro incorreto utilizado.
      - **6124: Estrutura do parâmetro**: detalhes estruturais do parâmetro utilizado está incorreto; por exemplo, tamanho, número de campos, subtipos.
      - **6125: Valores dos parâmetros**: o valor (numérico, lógico, caracteres) do parâmetro é incorreto.
      - **6126: Seqüência de parâmetros**: os parâmetros na seqüência de invocação estão em ordem incorreta, ou há muitos parâmetros, poucos parâmetros.
    - **613x: Retorno de Invocação de Componente**: relacionado à interpretação de parâmetros fornecidos pelo componente invocado ao retornar ao componente invocador. Neste contexto um registro, a seqüência de retorno de uma subrotina, ou um arquivo, podem se qualificar para esse tipo de defeito. Nota-se que os defeitos incluídos aqui *não* são defeitos nos componentes que criaram os dados de retorno mas sim na interpretação e manipulação subseqüente, feita pelo componente que recebeu os dados.

[3] Baseado no Apêndice "Bug Taxonomy and Statistics", SOFTWARE TESTING TECHNIQUES, segunda edição, por Boris Beizer. Direitos autorais© de Boris Beizer. Reimpresso com permissão de Van Nostrand Reinhold, Nova York.

- **6131: Identidade do parâmetro**: parâmetro de retorno incorreto acessado.
- **6132: Tipo do parâmetro**: tipo de parâmetro de retorno incorreto é usado; isto é, o componente usando o dado de retorno interpreta um parâmetro de retorno incorretamente, com relação ao tipo.
- **6134: Estrutura do parâmetro**: estrutura do parâmetro de retorno interpretada incorretamente.
- **6136: Seqüência de retorno**: a seqüência assumida para os parâmetros de retorno é incorreta.

* **614x: Iniciação de estado**: componente invocado não iniciado, ou iniciado no estado incorreto ou com dados incorretos.
* **615x: Invocação no lugar errado**: A posição onde o componente é invocado está incorreta.
* **616x: Invocação duplicada ou inútil**: o componente não deveria estar sendo invocado ou foi invocado mais vezes que o necessário.

– **62xx: INTERFACES EXTERNAS E TEMPORIZAÇÃO**: relacionado às interfaces externas, como dispositivos de entrada e saída e/ou controladores de dispositivos, ou outro software não operando sob a mesma estrutura de controle. Dados passados por arquivos ou mensagens estão qualificados para esse tipo de defeito.

* **621x: Interrupções**: defeitos relacionados à manipulação incorreta de interrupções ou preparação para interrupções; por exemplo, manipulador incorreto invocado, falha ao bloquear ou desbloquear interrupções.
* **622x: Dispositivos e Controladores**: interface incorreta com dispositivos ou controladores de dispositivos ou interpretação incorreta de dados sobre o status do dispositivo.
* **623x: Temporização e Vazão de E/S**: defeitos relacionados como a temporização e taxa de transferência de dados para dispositivos externos como: não atender requisitos de temporização definidos (demorou muito ou foi muito rápido), forçar a passagem de muitos dados, não aceitar taxa de transferência de dados como entrada.

Dos defeitos relacionados com a invocação de componentes (**611x**), o Teste de Integração teria condições de indicar a presença dos defeitos classificados como (**6112**) uso de componente incorreto; neste caso, tanto critérios baseados em fluxo de controle quanto os baseados em fluxo de dados seriam indicados. A própria construção do grafo de chamada permitiria a identificação desse tipo de problema e também dos defeitos classificados como (**616x**) duplicação ou ocorrência desnecessária de invocação a componente. Já o defeito (**6111**) componente invocado não existe – seria provavelmente detectado já no tempo de ligação do código.

Os defeitos classificados como (**612x**) relacionados aos parâmetros usados na invocação de um componente, (**613x**) relacionados ao retorno da invoção de componentes e (**614x**) envolvendo o estado e a iniciação dos componentes, podem ser detectados por critérios de teste de integração que exijam pelo menos a execução de toda chamada de subrotina no programa. Os Critérios Baseados em Análise de Fluxo de Dados têm, em geral, uma maior chance de identificar a presença desses defeitos, pois derivam os requisitos de teste através da análise dos parâmetros e das variáveis globais que definem as interfaces entre os

módulos, ao passo que os Critérios Baseados no Fluxo de Controle exigem simplesmente a execução de seqüências de chamadas no programa.

A invocação em lugar errado (**615x**) teria chance de ser detectada tanto no teste de integração quanto no teste de unidade, o uso de critérios estruturais facilita a detecção desse tipo de defeito.

Os defeitos classificados como (**62xx**) – interfaces externas e temporização – seriam mais facilmente detectados durante o teste de sistema, quando o software seria testado juntamente com todo o ambiente de operação (sensores, dispositivos de saída, controladores, etc.).

Leung e White [LW90] também apresentam uma descrição dos defeitos típicos de integração. Os autores dividem os defeitos em três categorias: i) Defeitos de Interpretação; ii) Defeitos na Chamada; e iii) Defeitos de Interface.

Defeitos de Interpretação: Argumenta-se que existem três especificações para um módulo: i) a *especificação documentada*, formada pela documentação de projeto; ii) a *especificação real* que corresponde ao comportamento do módulo (na realidade); e iii) a *especificação interpretada* como percebida por um usuário do módulo. Por simplicidade pode-se assumir que a especificação documentada é a mesma que a especificação real S_a do módulo, e deve ser a fonte usada para se julgar a correção no uso de um módulo. Um *defeito de interpretação* é um entendimento inadequado, por parte do usuário, de um módulo com respeito à S_a, de forma que ela difere da especificação interpretada S_i.

Como é difícil reconstruir S_i, o teste deve garantir que o usuário de um módulo não assumiu algo errado. Um problema comum em integrar módulos é que o tipo de comportamento de um módulo, esperado pelo usuário, pode não ser o mesmo que o efetivamente apresentado pelo módulo. Um defeito de interpretação ocorre sempre que a especificação interpretada diferir da especificação real. Pode-se categorizar defeitos de interpretação em três tipos – não se pode garantir que o teste de unidade vai revelar esses tipos de defeitos.

- *Defeito de função incorreta*: a funcionalidade implementada por um módulo chamado B pode não ser a requerida pela especificação do módulo chamador A. O programador pode assumir que B está implementando a função requerida por A.
- *Defeito de função extra*: dada a especificação dos módulos A e B, existe alguma funcionalidade de B não requerida por A, e existem algumas entradas de A que podem invocar essas funções e provocar uma falha em A.
- *Defeito de função ausente*: existem alguns valores de variáveis de A usados como entrada para B que estão fora da especificação de B; isso pode ser visto como B falhando em fornecer todas as funcionalidades requeridas por A.

Defeitos na Chamada: Um defeito na chamada é um defeito inserido por um engano no posicionamento da instrução de chamada, colocada num ponto incorreto do programa. Uma instrução de chamada é uma instrução que invoca a execução de um outro módulo. Uma sintaxe comum de uma instrução de chamada inclui o nome do módulo chamado, seguido pela lista de parâmetros reais. Esse defeito pode se manifestar de três maneiras diferentes: i) chamada extra – a instrução de chamada está em um caminho que não deveria conter a chamada; ii) chamada em lugar errado – a instrução de chamada está no lugar errado, mas no caminho que deveria conter a chamada; e iii) chamada ausente – a instrução de chamada está ausente no caminho que deveria conter a chamada.

Defeitos de Interface: Um defeito de interface ocorre sempre que o padrão de interface dos módulos é violado. Por exemplo, os parâmetros podem não estar na ordem correta, no

formato correto, ou no modo correto de entrada e saída. Um problema mais sério ocorre quando os domínios dos parâmetros reais e formais não combinam.

A principal contribuição de Leung e White em relação a Beizer está em classificar os Defeitos de Interpretação. A classificação e taxonomia de Beizer compreende aspectos relacionados à implementação em si; o aspecto psicológico de interpretação da especificação dos módulos no momento de realizar a integração não é capturado nessa classificação. Além dos Defeitos de Interpretação, Leung e White definem os Defeitos na Chamada; esses defeitos estão representados na taxonomia de Beizer pelo código (615X); por fim são classificados os Defeitos de Interface que na verdade, segundo a taxonomia de Beizer, incluem os Defeitos na Chamada e estão classificados com o código (6XXX).

A classificação dos defeitos típicos de integração é fundamental na definição de métodos e técnicas de teste; especialmente, para a definição de técnicas de teste baseadas em erro, como no caso da Análise de Mutantes [Del97]. Mesmo considerando-se técnicas estruturais, como a análise de fluxo de dados, o conhecimento das classes de defeitos comuns na integração do programa favorece a seleção de casos de teste que tenham maior probabilidade de revelar a presença desses defeitos, contribuindo de maneira essencial para o sucesso da atividade de teste.

2.5 Critérios para o Teste de Integração

O teste de integração é o teste aplicado quando todos os módulos individuais que compõem o software foram desenvolvidos e, em geral, testados. O teste é aplicado no nível de módulos e não mais no nível de comandos como no caso do teste de unidade. O teste de integração enfatiza a interação entre módulos e sua interface; conseqüentemente, a importância do teste de integração aumenta com a popularização da modularização no desenvolvimento de software [HS91].

2.5.1 Critérios de Linnenkugel e Müllerburg

Linnenkugel e Müllerburg [LM90] trabalharam na adaptação de critérios para o teste de unidade conhecidos na literatura para definir critérios semelhantes, baseados em análise de fluxo de controle e em análise de fluxo de dados, para o teste de integração de programas.

O modelo de integração utilizado representa um programa por um *grafo de chamada*. A abordagem concentra-se em analisar *relações* e *interfaces* entre módulos. As relações são determinadas pelos comandos de chamada nos módulos e as interfaces pelos dados usados por ambos, o *módulo chamador* e o *módulo chamado*. Para testar as relações entre os módulos foram adaptados Critérios baseados em Fluxo de Controle, similares a critérios do tipo Todos-Nós, Todos-Arcos e Todos-Caminhos, definidos para o teste de unidade. Para testar as interfaces as autoras adaptaram Critérios baseados em Fluxo de Dados, essencialmente a Família de Critérios Baseados em Fluxo de Dados de Rapps e Weyuker [RW85], apresentada na Seção 2.2.

Foi definido um conjunto de critérios baseados nos grafos de chamada, de maneira similar à definição dos critérios baseados em fluxo de controle, definidos sobre o grafo de programa (grafo de fluxo de controle).

Testando Relações, baseado no Grafo de Chamada:

> *Todos-Módulos* requer que todos os módulos do programa sejam executados pelo menos uma vez. O par (P, Ψ) satisfaz o critério *Todos-Módulos*, se e somente se, para todo módulo $m \in CG(P)$, existir pelo menos um caminho em Ψ no qual m ocorra.

Esse critério é análogo ao critério *Todos-Nós* definido para grafos de programas, no teste de unidade.

> *Todas-Relações* requer que as relações de chamada entre os módulos sejam exercitadas pelo menos uma vez. O par (P, Ψ) satisfaz o critério *Todas-Relações*, se e somente se, para todo multi-arco (M_1, M_2) em $CG(P)$, existir pelo menos um caminho em Ψ no qual um arco e de M_1 para M_2 ocorra.

Como o grafo de chamada é um multi-grafo pode haver mais de um arco entre dois módulos, correspondendo a mais de uma chamada de um módulo para outro; esse critério requer a execução de pelo menos um desses arcos para cada par de módulos conectados.

> *Todas-Relações-Múltiplas* requer que toda chamada entre módulos seja executada pelo menos uma vez. O par (P, Ψ) satisfaz o critério *Todas-Relações-Múltiplas*, se e somente se, para todo multi-arco (M_1, M_2) em $CG(P)$, e todo arco e de M_1 para M_2, existir pelos menos um caminho em Ψ no qual e ocorre.

Esse critério é análogo ao critério *Todos-Arcos* definido para o teste de unidade.

- *Todas-Seqüências-Simples-Descendente* ("All-Simple-Call-Sequences") requer que toda seqüência descendente de chamadas sem repetição de chamadas, seja executada pelo menos uma vez.
- *Toda-Seqüência-Descendente-Sem-Loop* ("All-Loop-Iteration-Call-Free-Sequences") requer que toda seqüência descendente de chamadas sem repetição de laços seja executada pelo menos uma vez.
- *Todas-Seqüências-de-Chamadas* ("All-Call-Sequences") requer que toda seqüência descendente de chamadas seja executada pelo menos uma vez.

Pode-se argumentar que esses critérios não são suficientes já que eles não requerem mais que a execução de chamadas, o que é insuficiente para testar a interface entre os módulos; ainda mais considerando-se que a interface entre dois módulos é determinada pelos dados usados por ambos. Linnenkugel e Müllerburg, motivadas por esse fato, aplicam os conceitos do teste baseado em análise de fluxo de dados para definir critérios para o teste de integração.

Testando a Interface

A interface entre os módulos é determinada pelos dados usados em ambos, o módulo chamador e o chamado. Esses dados, chamados *variáveis de comunicação*, podem ser tanto parâmetros como variáveis globais. Todas as outras variáveis não têm influência direta na

comunicação; portanto, durante o teste de integração não se tem interesse nessas variáveis. A seguir encontra-se a definição dos critérios, considera-se as definições de variáveis como sendo definições globais:

INT-Todas-Definições (*INT-All-Defs*) requer para toda chamada a execução de um sub-caminho livre de definição c.r.a cada variável de comunicação de cada definição para alguma referência (uso) dessa definição.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todas-Definições*, se e somente se, para cada chamada c do módulo M_1 para o módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x - Φ incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ onde p é livre de definição c.r.a x, $q \cdot (n)$ é livre de definição c.r.a $formal_c(x)$, e existe um c-uso$_{n'}(M_2, formal_c(x))$ global ou um p-uso$_{(n,n')}(M_2, formal_c(x))$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$ - Φ incluir pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$, onde p é livre de definição c.r.a $formal_c(x)$, $q \cdot (m)$ é livre de definição c.r.a x e existe um c-uso$_{m'}(M_1, x)$ global ou um p-uso$_{(m,m')}(M_1, x)$.

INT-Todos-C-Usos/Alguns-P-Usos (*INT-All-C-Uses/Some-P-Uses*) requer para toda chamada a execução de um sub-caminho livre de definição c.r.a cada variável de comunicação de cada definição relevante para todo c-uso. No caso de não haver um c-uso, um p-uso é requerido.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-C-Usos/Alguns-P-Usos*, se e somente se, para cada chamada c do módulo M_1 para o módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ para todo c-uso$_n(M_2, formal_c(x))$ global onde p é livre de definição c.r.a x e q é livre de definição c.r.a $formal_c(x)$; ou, no caso de não haver um c-uso, incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ para um p-uso$_{(n,n')}(M_2, formal_c(x))$ onde onde p é livre de definição c.r.a x e $q \cdot (n)$ é livre de definição c.r.a $formal_c(x)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ para todo c-uso$_m(M_1, x)$ global onde p é livre de definição c.r.a $formal_c(x)$ e q é livre de definição c.r.a x; ou, no caso de não haver um c-uso, incluir pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ para um p-uso$_{(m,m')}(M_1, x)$ onde onde p é livre de definição c.r.a $formal_c(x)$ e $q \cdot (m)$ é livre de definição c.r.a x.

INT-Todos-P-Usos/Alguns-C-Usos (*INT-All-P-Uses/Some-C-Uses*) requer para toda chamada a execução de um sub-caminho livre de definição c.r.a cada variável de comunicação de cada definição relevante para todo p-uso. No caso de não haver um p-uso, um c-uso é requerido.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-P-Usos/Alguns-C-Usos*, se e somente se, para cada chamada c do módulo M_1 para o módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir pelo menos um sub-caminho $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n, n')$ para todo p-uso$_{(n,n')}(M_2, formal_c(x))$ onde p é livre de definição c.r.a x e $q\cdot(n)$ é livre de definição c.r.a $formal_c(x)$; ou, no caso de não haver um p-uso, incluir pelo menos um sub-caminho $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n)$ para um c-uso$_n(M_2, formal_c(x))$ global onde p é livre de definição c.r.a x e q é livre de definição c.r.a $formal_c(x)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir pelo menos um sub-caminho $(n)\cdot p\cdot(n_{out}(M_2))\cdot q\cdot(m, m')$ para todo p-uso$_{(m,m')}$ (M_1, x) onde p é livre de definição c.r.a $formal_c(x)$ e $q\cdot(m)$ é livre de definição c.r.a x; ou, no caso de não haver um p-uso, incluir pelo menos um sub-caminho $(n)\cdot p\cdot(n_{out}(M_2))\cdot q\cdot(m)$ para um c-uso$_m(M_1, x)$ global onde p é livre de definição c.r.a $formal_c(x)$ e q é livre de definição c.r.a x.

INT-Todos-Usos (*INT-All-Uses*) requer para toda chamada a execução de um sub-caminho livre de definição c.r.a cada variável de comunicação de cada definição relevante para todo c-uso e todo p-uso.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-Usos*, se e somente se para cada chamada c do módulo M_1 ao módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir pelo menos um sub-caminho $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n)$ para todo c-uso$_n(M_2, formal_c(x))$ global onde p é livre de definição c.r.a x e q é livre de definição c.r.a $formal_c(x)$, e incluir pelo menos um sub-caminho $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n, n')$ para todo p-uso$_{(n,n')}(M_2, formal_c(x))$ onde onde p é livre de definição c.r.a x e $q\cdot(n)$ é livre de definição c.r.a $formal_c(x)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir pelo menos um sub-caminho $(n)\cdot p\cdot(n_{out}(M_2))\cdot q\cdot(m)$ para todo c-uso$_m(M_1, x)$ global onde p é livre de definição c.r.a $formal_c(x)$ e q é livre de definição c.r.a x, e incluir pelo menos um sub-caminho $(n)\cdot p\cdot(n_{out}(M_2))\cdot q\cdot(m, m')$ para todo p-uso$_{(m,m')}(M_1, x)$ onde onde p é livre de definição c.r.a $formal_c(x)$ e $q\cdot(m)$ é livre de definição c.r.a x.

INT-Todos-DU-Caminhos (*INT-All-DU-Paths*) requer para toda chamada a execução de todo DU-Caminho c.r.a cada variável de comunicação de cada definição relevante para todo c-uso e todo p-uso.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-DU-Caminhos*, se e somente se, para cada chamada c do módulo M_1 ao módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir todo sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ onde existe um c-uso$_{n'}(M_2, formal_c(x))$ e $(m) \cdot p \cdot (n_c)$ é um DU-Caminho c.r.a x e $(n_{in}(M_2)) \cdot q \cdot (n, n')$ é um DU-Caminho c.r.a $formal_c(x)$ ou onde existe um p-uso$_{(n,n')}(M_2, formal_c(x))$ e $(m) \cdot p \cdot (n_c)$ é um DU-Caminho c.r.a x e $(n_{in}(M_2)) \cdot q \cdot (n, n')$ é um DU-Caminho c.r.a $formal_c(x)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir todo sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ onde existe um c-uso$_{m'}(M_1, x)$ e $(n) \cdot p \cdot (n_{out}(M_2))$ é um DU-Caminho c.r.a $formal_c(x)$ e $q \cdot (m, m')$ é um DU-Caminho c.r.a x ou onde existe um p-uso$_{(m,m')}(M_1, x)$ e $(n) \cdot p \cdot (n_{out}(M_2))$ é um DU-Caminho c.r.a $formal_c(x)$ e $(n_c) \cdot q \cdot (m, m')$ é um DU-Caminho c.r.a x.

Silva [Sil94] também apresenta alguns critérios estruturais para o teste de integração baseados no grafo de chamada do programa. Os critérios – denominados todos-módulos, todos-arcos-chamadas e todos-caminhos – são os mesmos critérios definidos por Linnenkugel e Müllerburg como "All-Modules", "All-Multiple-Relations" e "All-Call-Sequences" para testar as relações entre os módulos. Silva apresenta também a ferramenta PROTESTE+ que apoia o teste estrutural para o teste de unidade, integração e de sistema.

2.5.2 Critérios de Harrold e Soffa

Harrold e Soffa [HS91] apresentam uma abordagem para o teste de integração que estende critérios de teste baseados em análise de fluxo de dados para o teste interprocedimental. É proposta uma técnica de análise que computa as associações definição uso interprocedimentais requeridas (para dependência direta e indireta) e um critério de teste que usa essa informação para selecionar e executar sub-caminhos pelas interfaces dos procedimentos.

O modelo de integração usado representa o fluxo de controle e de dados interprocedimental de um programa por um *grafo síntese* ("summary graph") [Cal88], onde nós representam regiões de código de interesse interprocedimental (entrada de procedimento, saída de procedimento, chamadas, retornos) e arcos representam informações de fluxo de controle.

O *grafo síntese* de um programa tem quatro tipos de nós: *nó de entrada*, *nó de chamada*, *nó de saída* e *nó de retorno*. Existem nós de entrada e nós de saída para cada *parâmetro formal* de todo módulo. Existem nós de chamada e nós de retorno para cada *parâmetro real* de todo ponto de chamada. O seguinte conjunto de regras é usado para se adicionar arcos ao grafo:

- são incluídos arcos do nó de chamada ao nó de entrada que correspondem à ligação de parâmetros formais aos parâmetros reais.
- São incluídos arcos do nó de saída para o nó de retorno que correspondem à mesma ligação.
- Um arco é adicionado sempre que uma definição de uma entrada alcança um ponto de chamada. A terminação do arco é o nó de chamada associado ao parâmetro real. O início do arco é o nó de entrada associado ao parâmetro formal.
- Um arco é adicionado sempre que uma definição de um nó de entrada alcançar um comando de retorno. A terminação do arco é o nó de saída associado ao parâmetro e o início é o nó de entrada associado ao parâmetro.
- Um arco é adicionado sempre que uma definição alcança, a partir de um ponto de chamada, um outro ponto de chamada ou comando de retorno. O início do arco é o nó de retorno associado ao parâmetro e a terminação é, ou o nó de chamada associado ao parâmetro real, ou o ponto de chamada, ou o nó de saída associado ao parâmetro.

A abordagem de Harrold e Soffa calcula as dependências de dados interprocedimentais com um método de quatro passos:

- Passo 1: Abstrair informação de fluxo de controle e de dados.
  - O primeiro passo é construir subgrafos de fluxo interprocedimentais tanto de fluxo de controle quanto de fluxo de dados, para cada procedimento. Nesses subgrafos os nós representam regiões de código que são de interesse interprocedimental, como entradas de procedimento, saída, chamada e retorno.
  - Determinar conjunto de usos consistindo da localização dos usos dos parâmetros formais que podem ser alcançados a partir do início do procedimento e a localização dos usos de parâmetros reais que podem ser alcançados a partir do retorno de uma chamada a procedimento.
  - Atribuir esses conjuntos de usos aos nós de entrada e nós de retorno, respectivamente.
- Passo 2: Representar o fluxo de controle e fluxo de dados interprocedimental.
  - Combinar os subgrafos obtidos no Passo 1, para criar o grafo de fluxo interprocedimental.
  - Adicionar arcos que representam a ligação dos parâmetros formais e reais tanto nos procedimentos chamadores quanto chamados.
  - Examinar, em cada ponto de chamada, a informação interprocedimental para determinar se o valor das variáveis permanecerá inalterado pela chamada. Caso isso seja verdade, um arco conecta a chamada ao nó de retorno correspondente.
- Passo 3: Obter informação interprocedimental.
  - Propagar a informação local nos nós do subgrafo pelo grafo de fluxo interprocedimental, em duas fases. Essas fases diferem no tipo de nó e arco percorridos.
    - 1ª **fase:** São propagadas as localizações de usos por referência de parâmetros que podem ser alcançados por chamadas de procedimentos.
    - 2ª **fase:** São propagadas as localizações de usos que podem ser alcançados por nós de retorno.

  - O resultado é um conjunto de usos alcançáveis para cada nó de chamada e de retorno do programa. Os conjuntos de usos alcançáveis são associados aos nós de chamada e de saída, já que esses nós representam pontos no programa onde o controle é alterado no nível interprocedimental.

- Passo 4: Computar informação definição-uso interprocedimental.
  - Utilizam-se tanto a informação local do Passo 1 quanto a informação propagada do Passo 3.
  - Considere cada definição de um parâmetro formal ou real e associe aos usos alcançáveis a partir dos pontos de definição.

Considere, como exemplo, o programa *Main* na Figura 2.3, e o grafo de fluxo dos módulos GetMax e PairMax na Figura 2.4. *GetMax* é um procedimento recursivo que lê o índice do primeiro e do último elemento de uma lista de, no máximo, 10 inteiros e retorna o maior valor da lista. Se somente o teste de unidade for aplicado sobre *GetMax*, os pares de definição e uso seriam: {MX, def={B_3, B_7}, uso={B_8}}, {M1, def={B_5}, uso={B_7}} e {M2, def={B_6}, uso={B_7}}.

```
1. program Main(input,output);

2. var  S:array[1..10] of integer;  (* processed lists up to 10 integers *)
3.      I, MAX, N: integer;

4.      procedure PairMax(var I,J,K:integer);  (* assigns maximum of I and J to K *)
5.      begin (* PairMax *)
6.              if I>J then K:=I
7.              else K:=J;
8.      end;  (* PairMax *)

9.      procedure GetMax(F,L:integer; var MX:integer);  (* gets the maximum of the list *)

10.     var M1, M2, MD: integer;
11.     begin (* GetMax *)
12.             if L<=F+1 then PairMax(S[F],S[L],MX)
13.             else begin
14.                     MD:=(F+L) DIV 2;
15.                     GetMax(F,MD,M1);
16.                     GetMax(MD+1,L,M2);
17.                     PairMax(M1,M2,MX);
18.             end;
19.     end; (* GetMax *)

20. begin (* Main *)
21.     readln(N);
22.     for I:= 1 to N do readln(S[I]);
23.     GetMax(1,N,MAX);
24.     writeln(MAX);
25. end. (* Main *)
```

Figura 2.3: Programa Exemplo de Harrold e Soffa.

A Figura 2.5 mostra o subgrafo de controle interprocedimental para o procedimento *PairMax* e um subgrafo parcial para o procedimento *GetMax*, representando as chamadas

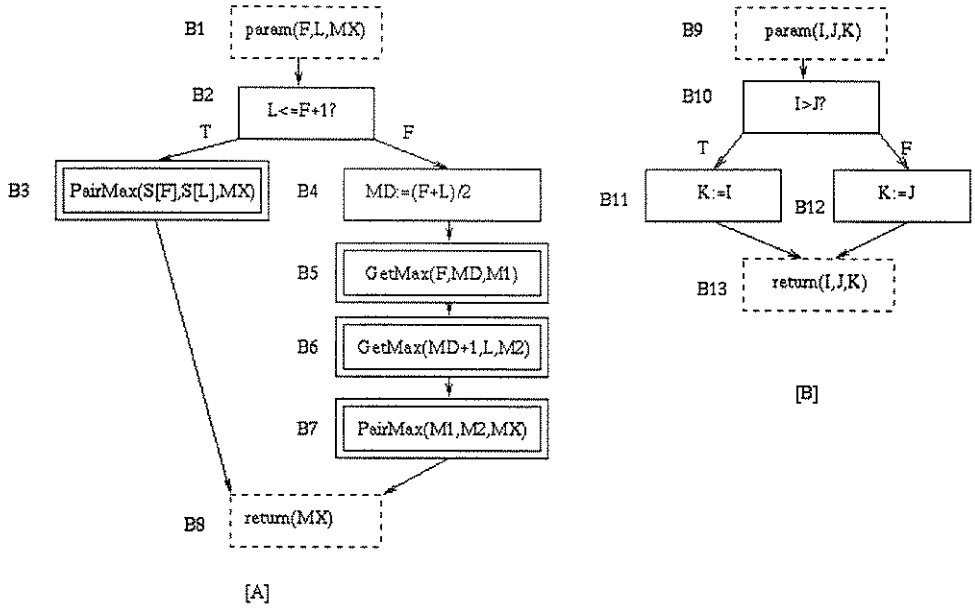

Figura 2.4: Grafo de Fluxo de *GetMax* e *PairMax*.

a procedimento nos nós B5, B6 e B7 da Figura 2.4. O resto do subgrafo de fluxo interprocedimental para *GetMax* representando a chamada a *PairMax* na linha 12 e o subgrafo interprocedimental para *Main* são similares.

No subgrafo de fluxo interprocedimental da Figura 2.5, os círculos representam chamadas e nós de retorno, círculos duplos representam nós de entrada e nós de saída, e linhas pontilhadas indicam que o fluxo de controle parte de um nó e alcança outro. Portanto, os nós 3, 5, 7, 9 e 11 são nós de chamada, o nó 3 representa o parâmetro real M1 na chamada recursiva à *GetMax* na linha 15 da Figura 2.4, o nó 5 representa o parâmetro real M2 na chamada recursiva a *GetMax* na linha 16, e os nós 7, 9 e 11 representam os parâmetros reais M1, M2 e MX na chamada a *PairMax* na linha 17. Os nós 4, 6, 8, 10 e 12 são os nós de retorno correspondendo aos nós de chamada 3, 5, 7, 9 e 11, respectivamente. Os pares de entrada e saída são {1,2}, {13,14}, {15,16} e {17,18}.

O arco (1,11) indica que o valor de MX que alcança a entrada de *GetMax* pode também alcançar a chamada a *PairMax*, onde é parâmetro real. Da mesma forma o arco (12,2) indica que um valor de MX que alcança um retorno de *PairMax* pode também alcançar uma saída de *GetMax*. O fluxo de controle em *PairMax* para os parâmetros formais I e J são representados pelos arcos (13,14) e (15,16), respectivamente.

Conjuntos de usos *U1*, *U2* e *U3* são associados aos nós de entrada e retorno e contém os usos dos parâmetros no procedimento que podem ser alcançados a partir do nó.

Os subgrafos de cada procedimento são então combinados. Adicionam-se arcos que representam o acoplamento entre parâmetros reais e formais, formando o grafo interprocedimental. A Figura 2.6 mostra o grafo para *GetMax* e *PairMax* com a inclusão desses arcos. Os arcos (7,13), (9,15), (11,17), (3,1) e (5,1) associam nós de chamada aos nós de entrada do procedimento chamado e os arcos (18,12), (16,10), (14,8), (2,4) e (2,6) associam os nós de saída do procedimento chamado com os nós de retorno no procedimento que faz a chamada. Arcos como (7,13) e (14,8) representam chamadas a outros procedimentos, enquanto arcos como (3,1) e (2,4) representam uma chamada recursiva. Arcos como (7,8) indicam que o parâmetro real M1 pode ser preservado numa chamada a *PairMax*.

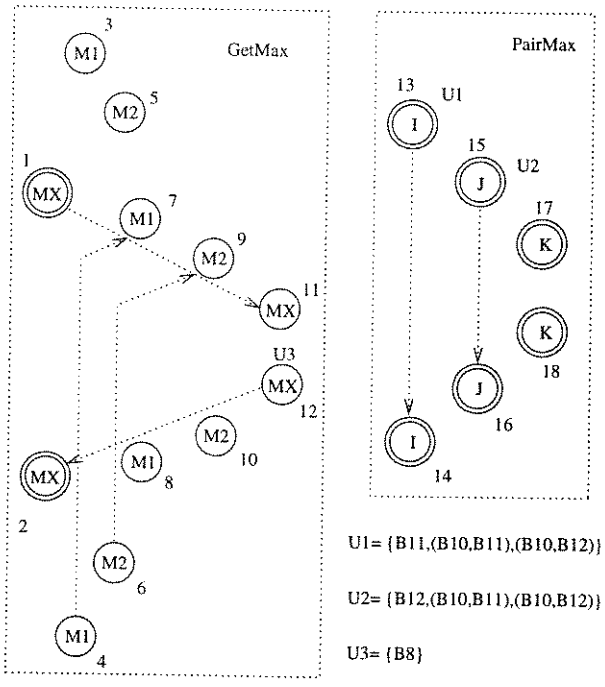

Figura 2.5: Passo Inicial na Construção do Grafo de Fluxo Interprocedimental para *GetMax* e *PairMax*.

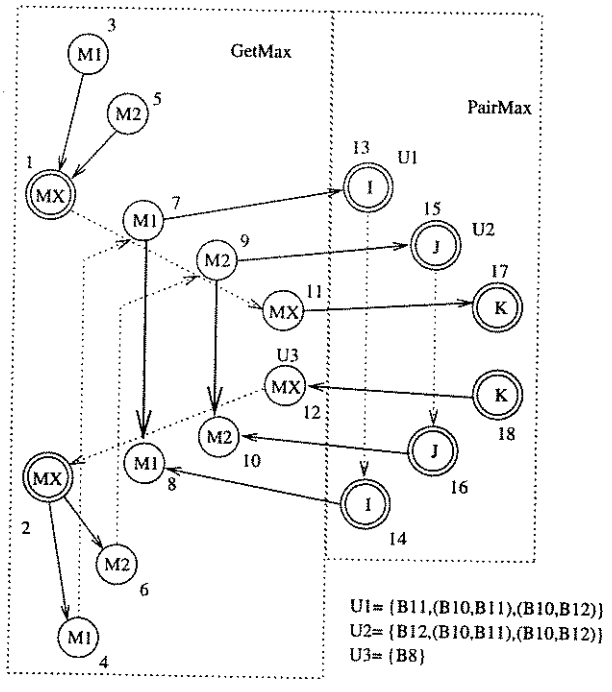

Figura 2.6: Grafo de Fluxo Interprocedimental para *GetMax* e *PairMax* com Acoplamento de Parâmetros Formais e Reais.

Em seguida, são associados aos nós de chamada de procedimentos e de saída conjuntos que representam os usos de parâmetros formais e de parâmetros reais que podem ser alcançados através dos limites dos procedimentos. Inicialmente, são propagadas as posições dos usos de parâmetros que podem ser alcançados através de uma chamada de procedimento. São então propagadas as posições dos usos que podem ser alcançados através de pontos de saída. Como resultado obtém-se para cada nó de chamada e de saída um conjunto de usos alcançáveis ("reachable-use sets – RU"). Na Figura 2.7, os conjuntos RU1 a RU4 são associados aos nós de chamada 3, 5, 7, 9 e 11 e aos nós de saída 18, 16, 14 e 2. Por exemplo, RU1 contém {B11,(B10,B11),(B10,B12)} é associado ao nó 7 e indica que esses usos de variável associados a M1 podem ser alcançados em *GetMax* pela chamada a *PairMax*.

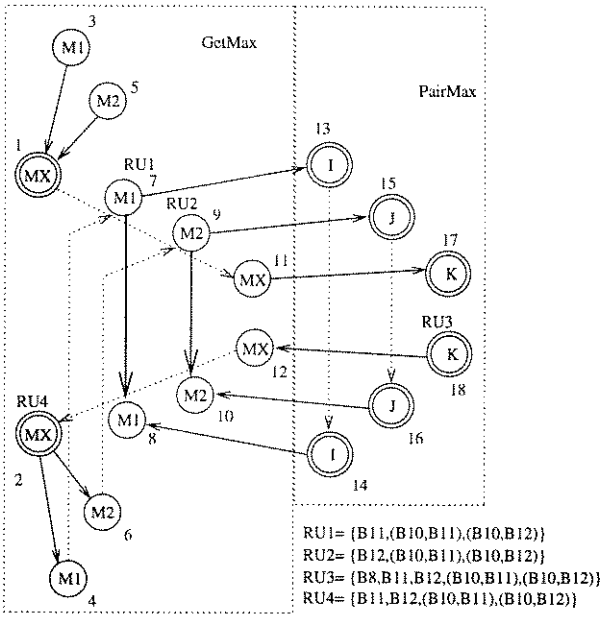

Figura 2.7: Grafo de Fluxo Interprocedimental para *GetMax* e *PairMax* com Conjunto de Usos Alcançáveis.

Finalmente, são calculadas as associações definição-uso interprocedimentais, utilizando-se tanto a informação de fluxo de dados local a cada procedimento como a informação interprocedimental. A Tabela 2.1 apresenta o resultado final para o exemplo da Figura 2.3. As definições de K em *PairMax* são os únicos pontos no programa onde a posição de memória associada a K em *PairMax*, MX em *GetMax*, e Max em *Main* é efetivamente modificada. Já que as duas definições de K em *PairMax* alcançam o final do procedimento, essas definições são associadas ao conjunto de usos alcançáveis RU3, associado ao nó de saída 18, formando assim as associações definição-uso interprocedimentais.

Tabela 2.1: Associações Definição-Uso Interprocedimentais entre *GetMax* e *PairMax*.

Variável	Definição	Uso
K em *PairMax*	B11	B8, B11, B12, (B10,B11), (B10,B12)
	B12	B8, B11, B12, (B10,B11), (B10,B12)

Basicamente, o critério proposto por Harrold e Soffa é o *INT-Todos-Usos* definido por Linnenkugel e Müllerburg; a diferença é a forma de modelar o fluxo de dados e o fluxo de controle do programa; ao invés de se representar o programa através de um grafo de

chamada, usa-se o grafo síntese. De qualquer forma, a informação sobre o fluxo de dados do programa tem que ser abstraída, logo de início, para construir o grafo síntese ou, mais tarde, para complementar o grafo de chamada.

2.6 Comparando Critérios de Teste

Dada a variedade de critérios de teste propostos na literatura, faz-se necessário um esforço no sentido de se definir formas de comparar esses critérios com o objetivo de disponibilizar informações sobre o custo, a força e a eficácia dos critérios. Vários estudos envolvendo a comparação entre critérios de teste vêm sendo realizados nos últimos anos; eles estão divididos em estudos teóricos e empíricos. Estudos empíricos têm sido conduzidos com o objetivo principal de investigar o custo de aplicação e a eficácia de diversos critérios de teste [Wey90, Wey88, BS89, WM95a]. Outros trabalhos apresentam comparações empíricas entre critérios incomparáveis sob o ponto de vista teórico [Won93, Sou96, Del97], como a análise de mutantes e os critérios baseados em fluxo de dados [WMM94, OPTZ96, SMV97, FWH97].

Segundo Wong e Mathur [WM95b], pode-se considerar três fatores na avaliação de um critério de teste. O *custo*, a *eficácia* e a *força*. O custo revela o esforço requerido para se satisfazer os requisitos de um critério de teste para um programa P. A eficácia é a probabilidade que um determinado critério tem de, satisfeitos seus requisitos, revelar a presença de pelo menos um defeito no programa. A *força* ou dificuldade de satisfação de um critério é determinada pela probabilidade de se satisfazer um critério dado que um outro foi satisfeito.

Do ponto de vista teórico, a *relação de inclusão* ("subsumption relation") [RW85, CPRZ89], a *análise de complexidade* [CPRZ85, RW85, Wey84, Nta88] e a *habilidade de detecção de defeitos* [FW93b, FW93a, FW93c] são consideradas para se avaliar, respectivamente, a força, o custo e a eficácia de critérios de teste.

A *análise de inclusão* estabelece uma hierarquia entre critérios fornecendo indicações com relação à força relativa de critérios de teste. A *análise de complexidade* fornece uma estimativa sobre o custo de aplicação de um critério no pior caso, considerando o número total de casos de teste requeridos para se satisfazer o critério. A *habilidade de detecção de defeitos* estabelece uma hierarquia entre critérios de teste, segundo sua eficácia em detectar defeitos, através da definição de relacionamentos que indicam que um critério tem uma maior probabilidade de detectar defeitos em um programa que outro; isto é, se um critério C_1 e um critério C_2 estiverem relacionados por $R(C_1, C_2)$, então C_1 é melhor em detectar defeitos que C_2.

Como foi definido na Seção 2.1, um critério de teste C_1 *inclui* um critério de teste C_2 se, para um grafo de fluxo de controle G, qualquer conjunto de caminhos de G que satisfaz C_1 também satisfaz C_2.

Considerando a *relação de inclusão*, um dos critérios baseados em análise de fluxo de dados mais exigente [TLK92], *Todos-DU-Caminhos* [RW85] é incluído por um dos Critérios Potenciais Usos; nenhum critério baseado em análise de fluxo de dados inclui os Critérios Potenciais Usos [Mal91] (Figura 2.8).

Rapps e Weyuker [RW85] mostram que seus critérios baseados em análise de fluxo de dados formam uma hierarquia de critérios segundo a relação de inclusão. A principal característica dessa hierarquia é a de promover uma ordem parcial entre o Critérios Todos-Arcos e

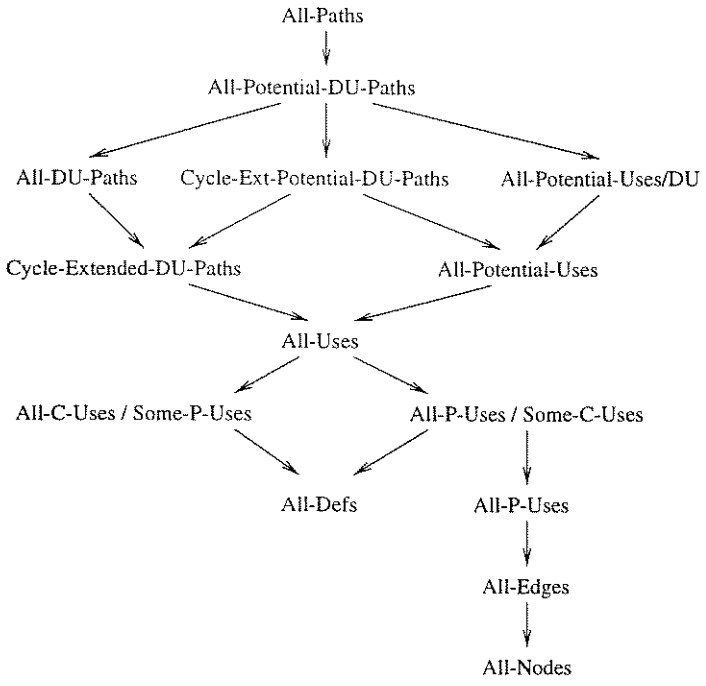

Figura 2.8: Relação de Inclusão.

o Critério Todos-Caminhos. Maldonado [Mal91] incluiu nessa hierarquia os Critérios Potenciais Usos, demonstrando que esses critérios incluem os critérios de fluxo de dados de Rapps e Weyuker, e mantêm a ordem parcial inicialmente estabelecida. Observa-se na Figura 2.8 duas das relações mais importantes nessa hierarquia de critérios, *Todos-Potenciais-DU-Caminhos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *Todos-DU-Caminhos* e *Todos-Potenciais-Usos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *Todos-Usos*, ambos trivialmente provados pela definição dos critérios já que todo *DU-Caminho* é também um *potencial-du-caminho* e todo *uso* é um *potencial uso*.

A análise de inclusão conduzida por Rapps e Weyuker [RW85] aplica-se a programas que satisfazem as seguintes suposições: cada predicado e cada comando deve fazer referência a pelo menos uma variável. Horgan e London [HL91] mostram que programas escritos em linguagem C, freqüentemente, violam essas suposições, invalidando a relação de inclusão. Em especial, Horgan e London demonstram que o Critério Todos-Usos deixa de incluir os Critérios Todos-Arcos e Todos-Nós quando programas escritos em C são considerados. Vergilio et al. [VMJ95] mostram que o Critério Todos-Potenciais-Usos inclui o Critério Todos-Arcos mesmo quando programas escritos em C são considerados. Essa constatação está refletida na hierarquia de critérios apresentada na Figura 2.9.

Além da relação de inclusão apresentada na Figura 2.8, Ntafos [Nta88] prova que os critérios K-tuplas requeridas *incluem* o Critério Todos-Usos e são incomparáveis com o Critérios Todos-DU-Caminhos; Ntafos também prova que os critérios de Laski e Korel *não incluem* o Critério Todos-Arcos. Ural e Yang [UY88] provam que, para um *programa bem formado*, o Critério Todos-OI-Caminhos Simples *inclui* o Critério Todos-DU-Caminhos.

A relação de inclusão apresentada na Figura 2.8 considera os Critérios Potenciais Usos básicos. 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 presença de caminhos não executáveis modifica consideravelmente algumas das propriedades dos critérios; por exemplo, Frankl [Fra87, FW88] observou que a relação

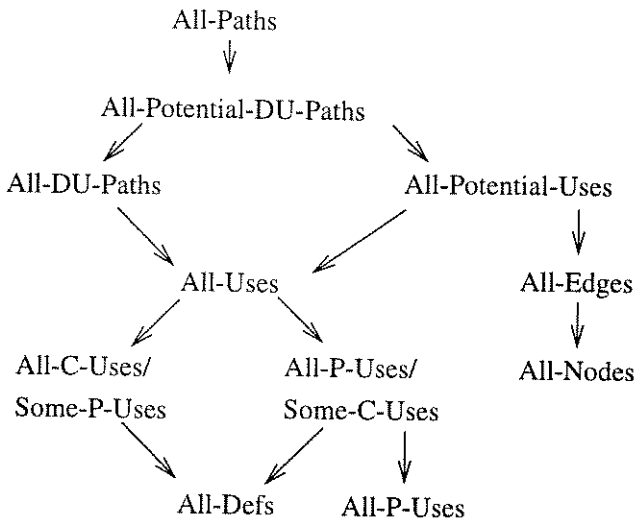

Figura 2.9: Relação de Inclusão Considerando a Linguagem C.

de inclusão entre os critérios da Família de Critérios de Fluxo de Dados muda significativamente. Maldonado [Mal91] apresenta a relação de inclusão para os Critérios Potenciais Usos e os critérios da Família de Critérios de Fluxo de Dados, essa relação é reproduzida na Figura 2.10. É importante notar que os Critérios Potenciais Usos preservam uma das características essenciais de um "bom" critério de teste, mesmo na presença de caminhos não executáveis: a de incluir o teste de ramos. Além disso, os Critérios Potenciais Usos são os únicos que preservam a hierarquia de critérios entre o teste de ramos e o teste de caminhos na presença de caminhos não executáveis, não sendo incluídos por nenhum outro critério baseado em análise de fluxo de dados.

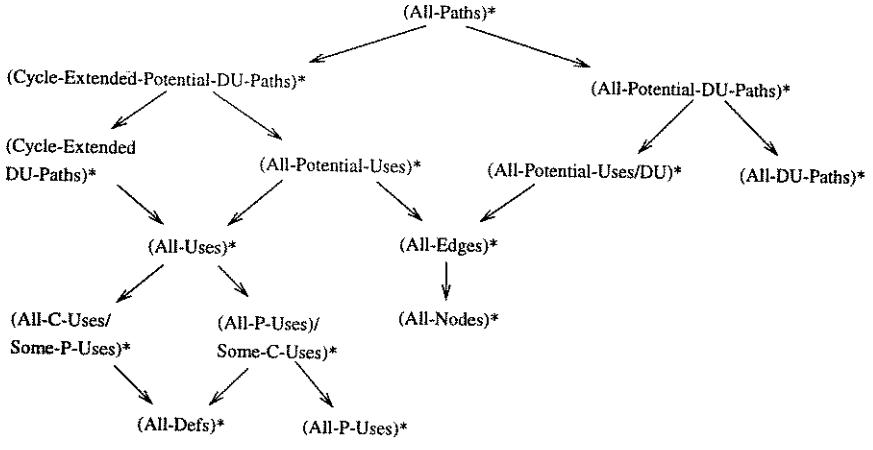

Figura 2.10: Relação de Inclusão para os Critérios Potenciais Usos e a Família de Critérios de Fluxo de Dados Executáveis.

A comparação em termos da relação de inclusão não considera explicitamente o custo de aplicação dos vários critérios, apesar de existir um indicação implícita desse custo. O número de casos de teste necessários para se satisfazer um critério é um fator que contribui para o custo de aplicação do critério; a análise de complexidade determina o número de casos de teste necessários, no pior caso, para se satisfazer cada critério [Nta88]. Maldonado [Mal91] mostrou que todo critério baseado em análise de fluxo de dados, mesmo o mais "fraco" de todos: *Todas-Definições*, tem complexidade maior ou igual a 2^t, sendo t o número de

comandos de decisão do programa. Com esse resultado Maldonado contesta o apresentado por Weyuker [Wey84], que estabelece a complexidade dos critérios da Família de Fluxo de Dados como sendo polinomial no número de comandos de decisão do programa.

Além disso, apesar de a relação de inclusão ser considerada quando se quer definir ou selecionar um critério, sabe-se que o fato de um critério C_1 incluir um critério C_2 não traz de forma explícita a informação sobre sua habilidade relativa de detecção de defeitos [FW93b].

Obviamente, uma das fraquezas dos critérios estruturais é que eles não são adequados a defeitos sensíveis a dados. Desta forma, não se pode dizer que um critério C_1 que inclui um critério C_2 é mais efetivo em revelar defeitos. De qualquer maneira, a ordem parcial estabelecida pela relação de inclusão pode ser usada para se definir estratégias de teste; por exemplo, como um guia para melhorar um dado conjunto de casos de teste T, de acordo com os objetivos e restrições envolvidas na atividade de teste.

A *habilidade de detecção de defeitos* é outra maneira de comparar critérios, estabelecendo a eficácia de um critério de teste. O objetivo é encontrar uma relação R entre um critério C_1 e um critério C_2 tal que, se $R(C_1, C_2)$ então pode-se garantir que C_1 é melhor que C_2 em detectar a presença de defeitos.

Frankl e Weyuker [FW93b] propuseram um conjunto de relações com o objetivo de comparar critérios de teste segundo sua relativa habilidade de detectar defeitos obtida através de medidas probabilisticas. Em outros trabalhos [FW93a, FW93c], essas autoras usaram as relações definidas para estabelecer uma hierarquia entre os critérios de teste mais conhecidos.

Muitas abordagens ao teste de software estão baseadas na idéia de se dividir o domínio de entrada do programa em sub-conjuntos, denominados subdomínios, para então requerer que o conjunto de teste inclua elementos de cada subdomínio. Esse tipo de estratégia é denominada de *Teste Baseado em Subdomínio.*

Em estratégias de subdomínio baseadas na estrutura do programa, cada subdomínio consiste de todos os elementos que provocam a execução de um dado fragmento de código. Por exemplo, no teste de ramos cada subdomínio consiste de todos os dados de entrada que causam a execução de um dado ramo.

Um critério de teste C é *baseado em subdomínio* se, para cada programa P e especificação S, existe um multi-conjunto não vazio $SD_C(P, S)$ de subdomínios (sub-conjuntos do domínio de entrada), de tal forma que C requer a seleção de um ou mais casos de teste de cada subdomínio.

É importante definir a estratégia de seleção de dados de teste usada para satisfazer um determinado critério de teste. As duas principais estratégias são: Sel_1) o testador seleciona casos de teste aleatoriamente até que o critério de teste tenha sido satisfeito; Sel_2) na segunda estratégia assume-se que o domínio de entrada foi dividido em subdomínios e então o testador seleciona aleatoriamente um número n, pré-determinado, de casos de teste de cada subdomínio. Assume-se que uma dessas duas estratégias de teste é usada. Na prática é raro se usar puramente uma das duas estratégias, usa-se na verdade uma combinação das duas, inicialmente seleciona-se um número de casos de teste para então verificar quais subdomínios ainda não foram satisfeitos e selecionar casos de teste específicos para satisfazê-los.

O conjunto de medidas da habilidade de detecção de defeitos está relacionado à probabilidade que um conjunto de teste tem de revelar pelo menos um defeito; três diferentes medidas são apresentadas. Considere um programa P, uma especificação S, e um critério C; sejam $d_i =\mid D_i \mid$ (cada D_i é um subdomínio do domínio de entrada de P) e m_i é o número

de entradas em D_i que causam falha (entradas que fazem com que a saída seja diferente daquela esperada segundo a especificação), onde $SD_C(P,S) = \{D_1, \cdots, D_k\}$.

Weyuker e Jeng [WJ91] notam que a habilidade de detecção de defeitos de um critério está relacionada com a concentração de entradas que causam falha no programa por seus subdomínios. A primeira métrica considerada é

$$M_1(C,P,S) = max_{1 \leq i \leq k}(\frac{m_i}{d_i})$$

Seja D_{max} o subdomínio com a maior concentração de entradas que causam falha, isto é, o subdomínio onde m_i/d_i é máximo. Considerando-se um critério baseado em subdomínio, qualquer conjunto de teste *C-adequado* contêm pelo menos um caso de teste de D_{max}. Usando-se Sel_1 ou Sel_2, cada elemento de D_{max} tem a mesma probabilidade de ser selecionado para "representar" D_{max}, e portanto M_1 é o limite inferior da probabilidade que um conjunto de teste *C-adequado* selecionado por Sel_1 ou Sel_2 tem de expor pelo menos um defeito.

$$M_2(C,P,S) = 1 - \prod_{i=1}^{k}(1 - \frac{m_i}{d_i})$$

M_2 é a probabilidade que um conjunto de teste selecionado por $Sel_2(1)$ (um elemento por subdomínio) tem de revelar pelo menos um defeito.

Um problema em se usar M_2 como uma métrica é que um critério baseado em subdomínio C_1 pode dividir o domínio D em k_1 subdomínios, enquanto outro critério C_2 divide D em k_2 subdomínios e $k_1 > K_2$. Desta forma M_2 concede a C_1 uma vantagem injusta já que C_1 requer k_1 casos de teste, enquanto C_2 requer apenas k_2; M_2 estaria comparando as qualidades em detectar defeitos de conjuntos de teste de tamanhos diferentes. Para corrijir esse problema define-se uma generalização da métrica M_2. Ao invés de sempre selecionar um caso de teste por subdomínio, n casos de teste são escolhidos. Essa métrica é chamada M_3:

$$M_3(C,P,S,n) = 1 - \prod_{i=1}^{k}[1 - (\frac{m_i}{d_i})]^n$$

A medida M_3 foi definida para ajustar a medida M_2 para critérios que determinam um número diferente de subdomínios; n é a razão entre o número de subdomínios definidos por um critério e o número de subdomínios definidos pelo outro, e representa o número de casos de teste a ser selecionado. A métrica M_3 pode ser ajustada para acomodar a diferença de tamanho dos conjuntos de casos de teste. Sejam k_1 e k_2 o número de subdomínios de C_1 e C_2, respectivamente, para um programa P e especificação S, e seja $r = (k_1/k_2)$, pode-se compensar a diferença no tamanho dos conjuntos de teste comparando-se $M_2(C_1,P,S) = M_3(C_1,P,S,1)$ com $M_3(C_2,P,S,r)$.

Cinco novas relações são definidas: *restringe* ("narrows"), *cobre* ("covers"), *particiona* ("partitions"), *cobre propriamente* ("properly covers") e *particiona propriamente* ("properly partitions"). O foco do estudo está em como cada critério subdivide o domínio de entrada em subdomínios. Considere o multiconjunto não vazio de subdomínios $SD_C(P,S)$ para um programa P, uma especificação S, e um critério C. Um critério $C_1 \overset{\text{restringe}}{\Longrightarrow} C_2$ para (P,S) se para todo subdomínio $D \in SD_{C_2}(P,S)$ existe um subdomínio $D' \in SD_{C_1}(P,S)$ tal que

$D' \subseteq D$. Frankl e Weyuker [FW93b] provaram que, desde que o critério C_1 e o critério C_2 requeiram a seleção de um caso de teste de cada subdomínio, a relação de inclusão é equivalente à relação restringe.

A Relação Cobre:

$C_1 \overset{\text{cobre}}{\Longrightarrow} C_2$ para (P, S) se para todo subdomínio $D \in SD_{C_2}(P, S)$ existe uma coleção não vazia de subdomínios $\{D_1, \cdots, D_n\}$, pertencentes a $SD_{C_1}(P, S)$ tal que $D_1 \cup \cdots \cup D_n = D$. C_1 *cobre universalmente* C_2 ("universally covers") se para todo par (P, S) - programa P, especificação S — $C_1 \overset{\text{cobre}}{\Longrightarrow} C_2$ para (P, S).

A Relação Particiona:

$C_1 \overset{\text{particiona}}{\Longrightarrow} C_2$ para (P, S) se para cada subdomínio $D \in SD_{C_2}(P, S)$ existe uma coleção não vazia $\{D_1, \cdots, D_n\}$ de subdomínios disjuntos dois a dois pertencentes a $SD_{C_1}(P, S)$ tal que $D_1 \cup \cdots \cup D_n = D$.

Relação Cobre Propriamente:

$C_1 \overset{\text{cobre propriamente}}{\Longrightarrow} C_2$ para (P, S) se existe um multi-conjunto $\mathcal{M} = \{D^1_{1,1}, \cdots, D^1_{1,k_1}, \cdots, D^1_{n,1}, \cdots, D^1_{n,k_n}\}$ tal que $\mathcal{M} \subseteq SD_{C_1}(P, S)$ e $D^2_1 = D^1_{1,1} \cup \cdots \cup D^1_{1,k_1}$ e, $\cdots$, $D^2_n = D^1_{n,1} \cup \cdots \cup D^1_{n,k_n}$. A diferença entre as relações *cobre* e *cobre propriamente* está em que na relação *cobre* cada subdomínio do critério C_2 deve ser a união de subdomínios de C_1 para $C_1 \overset{\text{cobre}}{\Longrightarrow} C_2$; e, para dizer que $C_1 \overset{\text{cobre propriamente}}{\Longrightarrow} C_2$, todos os subdomínios de C_1 usados para compor cada subdomínio de C_2 devem ser elementos distintos do multiconjunto $SD_{C_1}(P, S)$; se um subdomínio ocorre duas vezes nas uniões ele deve ocorrer duas vezes no multiconjunto.

A Relação Particiona Propriamente:

Se $C_1 \overset{\text{cobre propriamente}}{\Longrightarrow} C_2$ para (P, S) e cada coleção $D^2_1, \cdots, D^2_n$ é constituida de elementos disjuntos dois a dois, então $C_1 \overset{\text{particiona propriamente}}{\Longrightarrow} C_2$.

Frankl e Weyuker provaram que, considerando-se um dos métodos de seleção de casos de teste Sel_1 ou Sel_2, se um critério de teste C_1 particiona propriamente C_2, então $M_1(C_1) \geq M_1(C_2)$ e $M_2(C_1) \geq M_2(C_2)$; se C_1 particiona C_2, então $M_1(C_1) \geq M_1(C_2)$; e se C_1 cobre propriamente C_2, então $M_2(C_1) \geq M_2(C_2)$. A lista completa de relações e as conclusões das autoras quanto à habilidade de detecção de defeitos segundo as diversas medidas probabilísticas é apresentada na Tabela 2.2.

Tabela 2.2: Eficácia em Revelar Defeitos e as Relações de Frankl e Weyuker.

Relação	$M_1(C) \geq M_1(C')$	$M_2(C) \geq M_2(C')$	$M_3(C) \geq M_3(C')$
C restringe C'	nem sempre	nem sempre	nem sempre
C cobre C'	nem sempre	nem sempre	nem sempre
C particiona C'	sempre	nem sempre	nem sempre
C cobre propriamente C'	nem sempre	sempre	nem sempre
C particiona propriamente C'	sempre	sempre	nem sempre

No Capítulo 3 prova-se que os Critérios Potenciais Usos fazem parte de uma hierarquia de critérios entre os critérios *Todos-Arcos* e *Todos-Caminhos* mesmo quando a habilidade de deteção de defeitos está sendo considerada. No Capítulo 4 prova-se que esta relação é

mantida também para os Critérios Potenciais Usos de Integração.

2.7 Considerações Finais

Uma grande variedade de critérios de teste estruturais baseados em análise de fluxo de dados foi apresentada na literatura até o início dos anos 90. Hoje em dia a maior parte do esforço de pesquisa é dedicado à comparação, tanto teórica como empírica, desses critérios de teste. O objetivo é definir uma hierarquia de critérios que possa ser tomada como informação crucial no planejamento da atividade de teste.

A relação de inclusão foi muito usada na comparação entre critérios de teste; ela fornece uma maneira de se estabelecer uma hierarquia entre critérios de teste, constituindo uma informação essencial para o desenvolvedor de software. Mais recentemente, surgiu uma forma de comparar teoricamente critérios de teste segundo sua relativa habilidade de detectar defeitos. Frankl e Weyuker [FW93b] propuseram um conjunto de relações que têm a característica de ordenar os critérios de teste segundo sua capacidade de detectar defeitos. Essa nova forma de comparação complementa a análise de inclusão no sentido de deixar mais explícita a preocupação com a eficácia dos diversos critérios de teste.

No Capítulo 3 apresenta-se uma comparação entre a Família de Critérios de Fluxo de Dados [RW85] e os Critérios Potenciais Usos [Mal91] segundo sua habilidade de detectar defeitos. Essa análise é apresentada como uma extensão à análise de propriedades no nível do teste de unidade apresentada por Maldonado [Mal91] ao definir os Critérios Potenciais Usos.

Capítulo 3

Critérios Potenciais Usos: Análise e Extensões

Neste capítulo é apresentado um estudo sobre a aplicação dos Critérios Potenciais Usos para o teste de programas com uso extensivo de variáveis do tipo ponteiro, além de uma comparação com outras estratégias para se testar programas com esse tipo de variável. Na sequência é apresentada uma análise dos Critérios Potenciais Usos em termos de sua relativa habilidade para detecção de defeitos. O estudo sobre a aplicação dos critérios para programas com ponteiros e a análise da habilidade para detecção de defeitos encerram o estudo dos Critérios Potenciais Usos para o Teste de Unidade. A partir do próximo capítulo são tratados assuntos relativos ao Teste de Integração de programas.

3.1 Teste de Programas com Ponteiros

Testar um programa é geralmente uma tarefa difícil. O testador freqüentemente depara-se com uma dualidade: para garantir com um certo nível de confiança que o programa vai se comportar de acordo com o especificado, o testador tenta fazê-lo falhar. Por outro lado, sabe-se que a atividade de teste é incapaz de, no caso geral, demonstrar a correção de um programa. Para que se pudesse provar a correção de um programa através do teste, seria necessário demonstrar que ele executa corretamente para qualquer elemento do seu domínio de entrada, o que é inviável. Na prática, são selecionados do domínio de entrada subdomínios cujos elementos tenham um comportamento similar; o teste do programa resume-se em executar elementos representativos desses subdomínios.

O desafio é encontrar um critério de teste que possibilite subdividir o domínio de entrada de forma a minimizar o erro inerente a essa abordagem e, ao mesmo tempo, maximizar a probabilidade de se revelar a presença de um defeito ainda não revelado.

Dos critérios de teste definidos pode-se identificar, como já comentado, dois tipos básicos: os critérios funcionais e os critérios estruturais. Dos critérios estruturais destacam-se os critérios baseados em análise de fluxo de dados, preocupação central deste trabalho. Um dos aspectos principais dos critérios baseados em análise de fluxo de dados é a definição e identificação de associações de fluxo de dados, cada critério diferindo no modelo de fluxo de dados utilizado e na forma como as associações devem ser executadas. Uma fonte recente de preocupações é a influência da ocorrência de variáveis do tipo ponteiro na atividade de teste desses programas [LR90, HL91, OW91, LR92, PLR94, HR96, MF96, VMJ97b]. Em especial, preocupa-se com a adequação dos Critérios de Teste para o teste de programas com esse tipo de variável.

A ocorrência de variáveis do tipo ponteiro promove uma indecisão ao se definir quais

são as associações de fluxo de dados existentes em um programa. Como um ponteiro pode apontar para diferentes posições de memória em diferentes execuções de um mesmo comando, muitas vezes não se pode dizer com certeza, através de uma análise estática, qual variável está envolvida em um fluxo de dados específico. Uma abordagem para se determinar as associações de fluxo de dados em programas com ponteiros pode ser *não conservadora* se ela, mesmo correndo o risco de deixar de identificar algumas associações, procurar identificar de forma precisa as associações existentes. Uma abordagem é *conservadora* se ela procura identificar todas as associações de fluxo de dados que possam existir, mesmo que algumas dessas associações nunca ocorram quando o programa for executado. Esse problema é análogo ao associado ao uso de variáveis do tipo vetor ("arrays"); como o valor do índice, que indica a posição do vetor que está sendo acessada, geralmente só é conhecido dinamicamente, não se pode dizer, através de uma análise estática, qual posição do vetor está envolvida em um fluxo de dados específico.

Nesta seção é apresentada uma abordagem conservadora para a determinação de associações de fluxo de dados em programas com ponteiros. Essa abordagem está baseada no conceito de *potencial uso*, apresentado no capítulo anterior, e utiliza fundamentalmente a mesma abordagem adotada no modelo de fluxo de dados para variáveis compostas homogêneas [Mal91] usado na POKE-TOOL [Cha91] – ferramenta que implementa os Critérios Potenciais Usos para o Teste de Unidade – além de ser compatível com o tratamento adotado para vetores na POKE-TOOL. Além disso, são apresentados resultados da comparação entre a abordagem proposta e outras encontradas na literatura.

3.1.1 Conceituação

A ocorrência de ponteiros, vetores ("arrays") e estruturas de dados complexas, tipos de variáveis comuns em programas reais, dificultam a identificação de associações de fluxo de dados. Nos últimos anos diversos autores vêm trabalhando para resolver esse tipo de problema; em conseqüência disso, algumas abordagens já estão disponíveis. Ostrand e Weyuker [OW91] propõem um conjunto de critérios que analisam detalhadamente um programa com ponteiros na busca por associações de fluxo de dados. Horgan e London [HL91] propõem uma abordagem menos exigente que não trata separadamente todas as posições de memória que um ponteiro pode apontar; procura, ao invés disso, tratar essas posições de forma combinada para representar um elemento de fluxo de dados mais abstrato.

O cálculo das associações de fluxo de dados é complicado pela ocorrência de referências indiretas, sinônimos ("alias"), e chamadas a procedimentos, já que o verdadeiro endereço de memória referenciado por um identificador pode não ser conhecido em tempo de compilação. Se a ocorrência de ponteiros e outros tipos de variáveis que influenciam de maneira semelhante o fluxo de dados de um programa não for tratada, dois tipos de problemas podem ocorrer: *i*) algumas associações definição-uso em potencial não são detectadas pois a definição e o uso não são do mesmo objeto sintático, mesmo estando relacionados por uma referência de ponteiro; *ii*) um sub-caminho de uma definição para um uso de uma dada variável pode conter uma atribuição que cancela a definição original.

O objetivo da adequação de dados de teste baseada em análise de fluxo de dados continua o mesmo, não importando se a linguagem de programação tem ponteiros ou não. Entretanto, a análise estática nem sempre consegue determinar o conjunto exato de associações existentes em um programa que contenha ponteiros e/ou vetores. Tomando-se como exemplo a linguagem C, os tipos de dados fundamentais da linguagem (inteiro, caracter, etc.) podem ser referenciados através de dois tipos de ocorrências: *i*) a ocorrência de uma *variável*

de programa é qualquer ocorrência de variável cujo tipo é um dos tipos fundamentais; *ii*) uma ocorrência de *referência de ponteiro* é uma expressão dereferenciada do tipo ponteiro, como $*p$ ou $*(p+4)$. Além desses dois tipos de ocorrência que se referem ao armazenamento, existem as ocorrências das variáveis do tipo ponteiro: *iii*) uma *ocorrência de ponteiro* é a ocorrência de uma variável do tipo ponteiro. Nota-se que $*p$ inclui tanto a ocorrência de ponteiro quanto uma ocorrência de referência de ponteiro.

3.1.2 Trabalhos Anteriores

O trabalho de Rapps e Weyuker [RW85] na definição da Família de Critérios de Fluxo de Dados foi usado como base para uma série de outras pesquisas e investigações; algumas delas preocupadas com a precisão da determinação de associações de fluxo de dados.

Basicamente existem dois problemas envolvendo a determinação de associações de fluxo de dados: ponteiros e variáveis agregadas (compostas), como vetores. Na verdade esses dois problemas são similares e as soluções discutidas aqui para ponteiros, no caso dos Critérios Potenciais Usos, foram aplicadas no tratamento de variáveis compostas homogêneas [Mal91]. Outra solução específica para o tratamento de vetores foi proposta por Hamlet [HGN93] e considera a utilização de execução simbólica [Kin76] para tratar a ocorrência de cada elemento do vetor separadamente na busca por associações de fluxo de dados.

Um ponteiro é uma variável que armazena o endereço de outra variável. Conseqüentemente, para lidar com ponteiros deve-se considerar duas entidades distintas: o próprio ponteiro e a variável para a qual ele está apontando (às vezes chamada de variável dereferenciada). É em geral impossível saber, através de análise estática, para qual variável um ponteiro está apontando num dado ponto de execução de um programa. O conjunto de todas as variáveis para as quais o ponteiro pode estar apontando num determinado ponto do programa é chamado de *conjunto de sinônimos* do ponteiro ("alias set").

Abordagem de Ostrand e Weyuker

A Família de Critérios de Fluxo de Dados originalmente proposta por Rapps e Weyuker foi implementada para a linguagem Pascal na ferramenta ASSET [FW85]; essa ferramenta usa um modelo simples para lidar com ponteiros e variáveis agregadas. A proposta de Ostrand e Weyuker [OW91] foi estender a teoria apresentada por Rapps e Weyuker de forma que ela pudesse ser aplicada a linguagens com uso extensivo de ponteiros, como é o caso da linguagem C.

Apesar de a ferramenta ASSET aceitar praticamente qualquer programa Pascal sintaticamente correto, ela usa um modelo simplificado para manipular ponteiros e vetores. A extensão da teoria de análise de fluxo de dados para tratar especificamente de variáveis do tipo ponteiro deu origem à ferramenta TACTIC [OW91] ("Test Analysis and Coverage Tool, Intended for C").

A motivação por trás da análise de fluxo de dados é a de conferir se os casos de teste exercitam caminhos no programa nos quais existe fluxo de dados. Se uma variável recebe um valor em algum ponto do programa e nenhum caminho usando aquele valor é exercitado, é improvável que um defeito na atribuição do valor seja descoberto. Os critérios considerados por Ostrand e Weyuker, originalmente definidos por Rapps e Weyuker [RW85] e apresentados no Capítulo 2, requerem que os dados de teste de um programa causem a execução de sub-caminhos a partir da definição de uma variável (uma ocorrência onde a

variável recebe um valor) para um ou todos os seus usos (ocorrências onde o valor da variável é referenciado); ou seja, os pares definição-uso são o centro dessa análise de fluxo de dados.

Ostrand e Weyuker estenderam os conceitos de *definição* e *uso de variável* para incluir um modificador que representasse a certeza da existência da *definição* (ou do *uso*): os modificadores *de fato* ("definite") e *possível* ("possible"). Desta forma, se a análise estática de um programa determina de maneira não ambígua que uma variável do programa está sendo definida (ou usada), diz-se que existe uma definição *de fato* (ou um uso *de fato*). Mas se a análise estática não é capaz de determinar qual variável está sendo usada ou definida, o que pode ocorrer com ponteiros com *conjunto de sinônimos* com mais de uma variável, existe uma definição *possível* (ou um uso *possível*). Além disso, um *sub-caminho* $(n) \cdot k \cdot (m)$ é um *sub-caminho livre de definição de fato* c.r.a uma variável V se não existir nenhuma definição (de fato ou possível) da variável V em k. Se não existir nenhuma definição de fato e pelo menos uma definição possível da variável V o sub-caminho é dito *sub-caminho livre de definição possível* c.r.a variável V.

Qualquer ocorrência de um ponteiro p em uma expressão de dereferência é um uso de fato de p, já que o valor de p deve ser lido para se calcular a posição de memória a que $*p$ se refere. Desta forma, $*p$ e $*(p+5)$ são usos de fato de p. Uma atribuição para um ponteiro p, como no comando $(p = \&x;)$, é uma definição de p, mas *não* uma definição de $*p$. O comando $(*p = 17;)$ contém uma definição de $*p$, definições das variáveis no conjunto de sinônimos de p, e um uso de p.

Seguindo essas definições, Ostrand e Weyuker estabeleceram quatro níveis de associações definição-uso, classificadas de acordo com a certeza da existência de tais associações. Dada uma associação (n, m, V) onde n tem uma definição de V (de fato ou possível) e m tem um uso de V (também de fato ou possível), se a definição, o uso e todo sub-caminho livre de definição de n para m é um sub-caminho livre de definição de fato, c.r.a variável V, a associação é chamada *forte*. Se pelo menos um dos sub-caminhos livres de definição é possível, mas não todos, a associação é *firme*. Se todos os sub-caminhos livres de definição são possíveis, a associação é *fraca*. E se tanto a definição quanto o uso ou ambos forem possíveis a associação é *muito fraca*. Com isso definem-se quatro novos critérios baseados em análise de fluxo de dados como uma extensão do *Critério Todos-Usos*, são os critérios *Todos-Usos Forte*, *Todos-Usos Firme*, *Todos-Usos Fraco* e *Todos-Usos Muito Fraco*.

Numa associação *forte*, a definição e o uso são da mesma variável, existe pelo menos um sub-caminho livre de definição conhecido, e todos os sub-caminhos entre a definição e o uso são livres de definição ou com definição, mas sem ambiguidade. As associações *firmes* diferem das fortes já que existe pelo menos um sub-caminho da definição para o uso cuja ausência de definição não pode ser garantida em tempo de execução, além de pelo menos um sub-caminho sabidamente livre de definição. Nas associações *fracas*, nenhum dos sub-caminhos é garantidamente livre de definição, mas ainda se sabe que a definição e o uso dizem respeito à mesma variável. Finalmente, nas associações *muito fracas*, tanto o uso como a definição, ou ambos, são referências de ponteiro que podem corresponder a mais de uma variável e podem não ser as mesmas.

A Figura 3.1 mostra um programa apresentado como exemplo por Ostrand e Weyuker [OW91]; o programa foi construído apenas para ilustrar as idéias exploradas pelos autores, não pretende executar nenhuma tarefa específica, e nem mesmo ser um exemplo de uma prática comum ou aconselhável de programação. De qualquer forma, o mesmo exemplo será usado nesta seção para que se possa apontar as diferenças entre as abordagens analisadas. A lista de associações encontradas por Ostrand e Weyuker nesse programa são apresentadas na Tabela 3.1.

```
    proc(condition1, condition2)
    int condition1, condition2;
    {
        int x, y, z, *p;
24:     z = 17;
25:     x = 13;
26:     if (condition1) {
28:         p = &y;
29:         *p = z;
        }
        else {
35:         if (condition2)
36:             p = &x;
37:         else
38:             p = &z;
40:         *p = 7 + z;
42:         y = 53;
43:         p = &x;
        }
49:     x = x + y + z;
50:     *p = *p + 5;
51:     y = x + y;
    }
```

Figura 3.1: Programa Exemplo de Ostrand-Weyuker para Análise de Ponteiros

A definição de z na linha 24 participa de três associações: (24,29,z) e (24,40,z) são fortes, enquanto (24,49,z) é firme, pois os dois sub-caminhos deixando o ramo falso da *condição1* ("condition1") passam pelo comando (40: *p=7+z;). Como o conjunto de sinônimos de $*p$ neste ponto é $\{x, z\}$, esse comando é uma possível definição de z. A definição de y na linha 42 participa de três associações. A associação (42,49,y) é forte, já que o sub-caminho da definição ao uso é livre de definição c.r.a y; (42,51,y) é fraca, já que o único sub-caminho da definição para o uso passa pelo comando 50, onde y é incluído no conjunto de sinônimos de $*p$; finalmente (42,50,y) é uma associação muito fraca, já que o uso na linha 50 é um uso possível de y, como um sinônimo de $*p$.

A informação analisada até agora é totalmente abstraída da análise estática do programa. Em tempo de execução, a preocupação central é determinar quais associações, definidas estaticamente, são efetivamente exercitadas quando o programa é executado por um dado de teste. Uma associação (n,m,V) é considerada executada por um dado de teste t se: i) a execução de t faz com que o controle chegue ao nó n onde um valor é atribuído à variável V; ii) o controle eventualmente alcança o nó m, onde V é referenciada e seu valor é usado numa computação ou na avaliação de um predicado, e iii) o valor de V não é modificado em nenhum comando executado depois que o valor é atribuído a V em n até que o valor seja usado em V.

Essa abordagem, apresentada por Ostrand e Weyuker, além de exigir o cálculo do conjunto de sinônimos para cada ocorrência de ponteiro no programa, requer um monitoramento em tempo de execução para se determinar qual associação foi efetivamente executada. Além disso, ela considera apenas o uso de ponteiros como uma variável escalar, quando o mais comum é termos o uso de ponteiros associados a vetores; neste caso, a maioria das associações identificadas por essa abordagem seria "muito fraca" já que tanto o conjunto de

sinônimos do ponteiro como o índice do vetor estariam colaborando para a não precisão no cálculo das associações.

Uma proposta para eliminar o custo adicional introduzido pelo monitoramento em tempo de execução foi apresentada por Marx e Frankl [MF96]; a idéia é substituir o cálculo dos sinônimos através de uma abordagem orientada a caminhos e não orientada a um ponto do programa, como na proposta de Ostrand e Weyuker. Nessa proposta os sub-caminhos entre uma definição e um uso são representados por expressões regulares; desta forma, é possível saber estaticamente para qual variável um ponteiro estará apontando, dado que a execução passou pelo sub-caminho representado. Essa abordagem restringe e altera a definição original dos critérios baseados em fluxo de dados que exigem a cobertura de associações definição-uso; na definição original não é requerido nenhum sub-caminho específico para cobrir uma associação, além de inserir uma sobrecarga adicional para se calcular as expressões regulares que representam os diversos sub-caminhos entre uma definição e um uso dentro de um programa.

Abordagem de Horgan e London

Horgan e London [HL91] apontaram algumas das dificuldades de se aplicar análise de fluxo de dados em programas escritos em C, baseados em sua experiência no desenvolvimento da ferramenta ATAC – ferramenta para análise de cobertura baseada em fluxo de dados para programas escritos em C.

O problema de ponteiros é também discutido nesse trabalho, mas a abordagem é substancialmente diferente daquela adotada por Ostrand e Weyuker. Horgan e London primeiro definiram uma *variável de fluxo de dados* como sendo: um nome que se refere a uma posição de memória alocada estaticamente ou todas as expressões que envolvem uma variável dereferenciada como um ponteiro. Mas, ao invés de considerar cada ponteiro com seu conjunto de sinônimos e estabelecer níveis de associações dependendo da certeza que se tem sobre a existência de tais associações, eles apenas consideram cada ocorrência de uma variável do tipo ponteiro como a ocorrência de um objeto de dados integrado ou uma "variável de fluxo de dados composta"; desta forma $*p$, $(**p)$, $*(p+10)$ e $p[i]$ se referem ao mesmo objeto de dados.

O custo de aplicação dessa abordagem é aparentemente menor do que o da proposta por Ostrand e Weyuker. O problema é que Horgan e London usam a mesma família de critérios definida por Rapps e Weyuker [RW85] e, conseqüentemente, continuam procurando pela ocorrência de uma definição de variável e pela ocorrência explícita de um uso para estabelecer uma associação de fluxo de dados; isso leva a uma depreciação nos requisitos de teste (medida pelo número de associações requeridas) (Tabela 3.1).

3.1.3 Abordagem Potenciais Usos

Os *Critérios Potenciais Usos* [Mal91] diferem da *Família de Critérios de Fluxo de Dados* definida por Rapps e Weyuker [RW85] fundamentalmente porque eles não requerem a ocorrência explícita do *uso* de uma variável para estabelecer uma associação de fluxo de dados. Desta forma, se a definição de uma variável está viva em um certo ponto do programa e poderia estar sendo usada naquele ponto, um *potencial uso* existe – e estabelece-se uma *potencial associação* entre o ponto onde uma variável é definida e todos os pontos alcançáveis por sub-caminhos livres de definição.

Outro ponto fundamental, usado na definição da abordagem para tratamento de ponteiros, é o *Modelo de Fluxo de Dados* usado na implementação dos Critérios Potenciais Usos. As ocorrências de variáveis em um programa podem ser de três tipos: uma *definição*, um *uso* ou uma *indefinição*. Uma *definição* ocorre quando um valor é atribuído a uma variável. Um *uso* ocorre quando o valor da variável é acessado. Uma *indefinição* ocorre quando a variável não está associada a uma posição de memória ou seu valor não está acessível.

Os Critérios Potenciais Usos requerem somente que cada nó i no grafo de programa seja associado com o conjunto de variáveis definidas naquele nó. A versão do grafo de programa estendida com informações de fluxo de dados é chamada *grafo-def*.

Geralmente 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) numa chamada de procedimento como parâmetro de saída. Um parâmetro numa chamada de procedimento pode ser de três tipos: valor, referência ou nome [GJ87]. Se a variável é usada como referência ou nome ela é considerada um parâmetro de saída.

As definições que podem ocorrer devido a uma chamada de procedimento são diferenciadas das outras e são chamadas *definições por referência* [Mal91]. As definições por referência são as grandes responsáveis pela imprecisão na determinação das associações de fluxo de dados, já que é, em geral, impossível saber se uma variável foi ou não definida numa chamada de procedimento. Uma abordagem conservadora é considerar que a variável é definida naquele ponto, mas não considerá-la uma redefinição de uma variável definida anteriormente. Essa abordagem também é utilizada no tratamento de variáveis do tipo vetor.

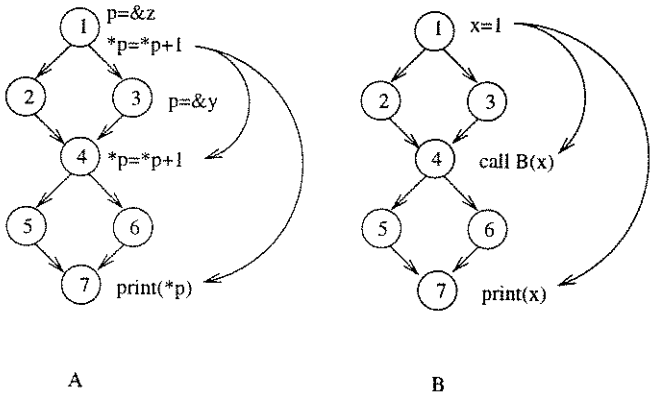

Figura 3.2: Tratamento de Ponteiros X Definição por Referência

O que se pretende agora é usar a mesma abordagem quando consideram-se variáveis do tipo ponteiro; na verdade, a abordagem propõe combinar os Critérios Potenciais Usos através do estabelecimento de associações de fluxo de dados baseadas no conceito de potencial uso com a mesma estratégia usada no modelo de fluxo de dados para variáveis definidas por referência em chamadas de procedimento e variáveis do tipo vetor. A Figura 3.2 ilustra como a abordagem funciona; na Figura 3.2-B tem-se a definição da variável x no nó 1 e a definição por referência de x na chamada ao procedimento B no nó 4. A estratégia conservadora não usa essa definição por referência no nó 4 para impedir a busca de associações da definição de x no nó 1 para os potenciais usos nos nós 5, 6 e 7; por exemplo, a definição de x no nó 1 e o seu potencial uso no nó 7 dariam origem à potencial associação $< 1, 7, \{x\} >$. Obviamente ainda existiriam as associação iniciando na definição de x (definição por referência) no nó 4, dando origem, por exemplo, à associação $< 4, 7, \{x\} >$. Pode-se dizer que a definição por referência

da variável x no nó 4 é considerada uma definição de variável, mas não é considerada uma redefinição de variável; a abordagem é conservadora pois não se pode garantir que a variável x foi redefinida na chamada ao procedimento B. Da mesma forma, a Figura 3.2-A ilustra a mesma idéia no tratamento de variáveis do tipo ponteiro, já que não se pode garantir que a definição de $*p$ no nó 4 está redefinindo a variável $*p$ – correspondendo à variável z – definida no nó 1; desta forma, a abordagem conservadora não considera essa definição como sendo uma redefinição da variável $*p$ definida no nó 1.

Considere o grafo de programa da Figura 3.3, correspondente ao programa exemplo apresentado na Figura 3.1 para que se possam analisar algumas das diferenças entre as abordagens.

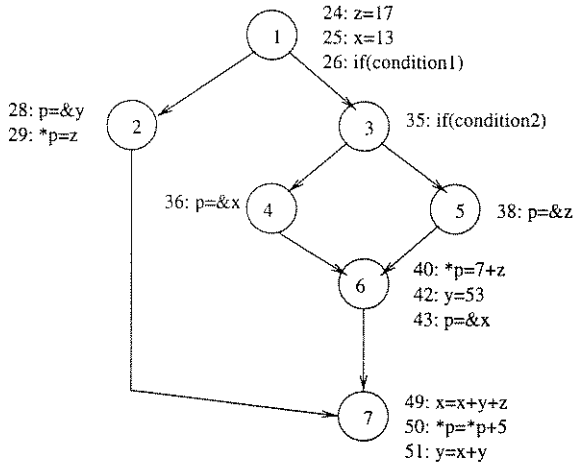

Figura 3.3: Grafo de Programa – Exemplo de Ostrand e Weyuker

A lista completa de associações identificadas por Ostrand e Weyuker (O-W), Horgan e London (H-L), e uma lista parcial das associações identificadas pela abordagem Potencial Uso (P-U) no programa da Figura 3.3 são apresentadas na Tabela 3.1. Adicionalmente a abordagem Potencial Uso identifica as associações apresentadas na Tabela 3.2.

Analisando a Tabela 3.1 e a Figura 3.3 pode-se identificar 6 situações de interesse:

Situação 1 - Associações 3, 16 e 21: Nessa situação, existe uma *definição* de uma variável do tipo ponteiro e um subsequente *uso* de uma variável no conjunto de sinônimos do ponteiro. Ou o contrário, uma *definição* de uma variável e um *uso* do ponteiro.
O-W: Nessa abordagem, identifica-se a associação calculando-se o conjunto de sinônimos do ponteiro.
H-L: Não identifica a associação.
P-U: Identifica a associação já que existe um *sub-caminho livre de definição* com relação ao ponteiro ou com relação à variável.

Situação 2 - Associação 12: Ostrand e Weyuker a consideram uma associação ilegítima, já que $*p$ é redefinida no comando 50.
O-W: Requer a associação pois o conjunto de sinônimos de p no comando 50 é $\{x,y,z\}$ e por essa abordagem não é possível identificar a redefinição de y dereferenciado por $*p$.
H-L: Não requer a associação pois procura pela ocorrência explícita do uso do ponteiro.
P-U: Requer a associação devido ao Modelo de Fluxo de Dados usado com os Critérios

Tabela 3.1: Lista de Associações Requeridas

#	O-W		H-L	Pot-Uses	
1	Strong	(24,29,z)	(24,29,z)	(24,29,z)	
2		(24,40,z)	(24,40,z)	(24,40,z)	
3		(29,49,y)	—	(29,49,$*p$)	
4		(42,49,y)	(42,49,y)	(42,49,y)	
5		(28,29,p)	(28,29,p)	(28,29,p)	
6		(28,50,p)	(28,50,p)	(28,50,p)	
7		(36,40,p)	(36,40,p)	(36,40,p)	
8		(38,40,p)	(38,40,p)	(38,40,p)	
9		(43,50,p)	(43,50,p)	(43,50,p)	
10	Firm	(24,49,z)	(24,49,z)	(24,49,z)	
11		(25,49,x)	(25,49,x)	(25,49,x)	
12	Weak	(29,51,y)	—	(29,51,$*p$)	
13		(42,51,y)	(42,51,y)	(42,51,y)	
14		(49,51,x)	(49,51,x)	(49,51,x)	
15	Very Weak	(29,50,y)	(29,50,$*p$)	(29,50,$*p$)	
16		(42,50,y)	—	(42,50,y)	
17		(50,51,y)	—	(28,51,p)	(50,51,$*p$)
18		(50,51,x)	—	(43,51,p)	
19		(40,49,z)	—	—	(40,49,$*p$)
20		(40,49,x)	—	—	
21		(49,50,x)	—	(49,50,x)	
22		—	(40,50,$*p$)	(40,50,$*p$)	

Potenciais Usos, a redefinição de um ponteiro não faz com que se pare de procurar por potenciais associações.

Situação 3 - Associação 13: É classificada como *fraca* por O-W; mas, como os próprios autores apontam, essa é uma associação *forte* já que $*p$ em 50 nunca apontaria para y pela execução do comando 42. H-L e P-U classificam essa associação no mesmo nível que qualquer outra.

Situação 4 - Associação 15: *definição* e *uso* de uma variável do tipo ponteiro.
O-W: Requer uma associação para cada elemento do conjunto de sinônimos do ponteiro. Neste exemplo só existe um elemento $\{y\}$
H-L & P-U: Somente uma associação é requerida.

Situação 5 - Associações 17, 18, 19 e 20: *Definição* de um ponteiro e *uso* de uma variável do conjunto de sinônimos.
O-W: Requer uma associação para cada elemento do conjunto de sinônimos do ponteiro.
H-L: Não requer a associação pois procura pela ocorrência explícita do uso do ponteiro.
P-U: Uma associação é requerida a partir da *definição* do ponteiro até o *uso* da variável no conjunto de sinônimos. Além disso, também será requerida uma associação a partir de cada ponto onde o ponteiro passou a apontar para uma variável ao

ponto onde essa variável está sendo usada, associações {17,18}.

Situação 6 - Associação 22: *Definição* de $*p$, redefinição de p e *uso* de $*p$. O-W não requer essa associação devido à redefinição de $*p$ em 43.

Observa-se que a abordagem de Horgan e London, apesar de requerer um número menor de associações de fluxo de dados e, conseqüentemente, ter condições de apresentar um custo de aplicação menor que os das outras abordagens, não exige várias associações importantes. A abordagem Potenciais Usos tem, a princípio, um custo menor que a abordagem de Ostrand e Weyuker pois não requer que se calcule o conjunto de sinônimos para cada variável do tipo ponteiro do programa e não estabelece a necessidade de se ter um monitoramento do programa em tempo de execução. E, apesar de ter potencialmente um custo de aplicação maior que o da abordagem de Horgan e London, ao superar as outras abordagens em número de elementos requeridos, torna o teste de software mais eficaz no objetivo de identificar a presença de defeitos no programa.

Tabela 3.2: Associações Potenciais Usos Adicionais

Additional Potential Uses Associations					
23	(24,25,z)	24	(24,26,z)	25	(24,28,z)
26	(24,50,z)	27	(24,51,z)	28	(24,35,z)
29	(24,36,z)	30	(24,38,z)	31	(24,42,z)
32	(24,43,z)	33	(25,26,x)	34	(25,28,x)
35	(25,29,x)	36	(25,35,x)	37	(25,36,x)
38	(25,38,x)	39	(25,40,x)	40	(25,42,x)
41	(25,43,x)	42	(28,49,p)	43	(36,42,p)
44	(36,43,p)	45	(38,42,p)	46	(38,43,p)
47	(40,42,$*p$)	48	(40,43,$*p$)	49	(40,51,$*p$)
50	(42,43,y)	51	(43,49,p)		—

A abordagem Potencial Uso ainda requer o conjunto de 29 associações listadas na Tabela 3.2. Algumas das características interessantes desse conjunto de associações podem ser observadas: as associações 29 e 30 mostram que para uma única definição de z os ramos "verdadeiro" e "falso" do comando if no nó 35 estão sendo requeridos (a mesma situação ocorre com as associações 37 e 38 para a variável x); as associações 23, 31, 40, 43, 45 e 47 conectam a definição de uma variável ao ponto onde outra variável está sendo definida; esta característica não está presente em nenhum dos critérios discutidos anteriormente e, geralmente, só é conseguida através do uso de técnicas baseadas em erro; por exemplo, a Análise de Mutantes [DLS78].

Análise de Inclusão

Na discussão que se segue as definições de *forte*, *firme*, *fraco* e *muito fraco* são apresentadas exatamente como definidas por Ostrand e Weyuker, para se fazer uma comparação com a abordagem Potenciais Usos em termos da relação de inclusão.

Considere-se inicialmente o seguinte teorema:

Teorema 3.1 (Sub-caminhos livres de definição) *Considere uma associação de fluxo de dados (n,m,V). Todo* sub-caminho livre de definição de fato $(m) \cdot k \cdot (n)$ *onde m tem uma* definição *de variável e n tem um* uso *da mesma variável é um* sub-caminho livre de definição *no Modelo de Fluxo de Dados dos Critérios Potenciais Usos.*

Prova: Deve-se analisar dois casos,

Caso 1 : m tem uma *definição* da variável z (ou um ponteiro p); já que $(m) \cdot k \cdot (n)$ é um sub-caminho livre de definição de fato, não há *redefinição* de z (ou p) em k; conseqüentemente $(m) \cdot k \cdot (n)$ é um *sub-caminho livre de definição* no Modelo de Fluxo de Dados dos Critérios Potenciais Usos.

Caso 2 : m tem uma *definição* de uma variável dereferenciada $*p$; neste caso duas situações podem ocorrer: i) não há redefinição de $*p$ no sub-caminho; ou ii) existe uma *redefinição* de $*p$ mas o conjunto de sinônimos dessa ocorrência é completamente diferente do conjunto em n. Na situação i o sub-caminho é obviamente um *sub-caminho livre de definição*; na situação ii o sub-caminho é considerado *livre de definição* de acordo com o Modelo de Fluxo de Dados, ignorando-se a *redefinição* de $*p$.

Seguem-se as definições das associações *forte*, *firme*, *fraca* e *muito fraca*. Para cada uma das definições apresentadas, é incluída uma discussão sobre o comportamento das potenciais associações definição-uso. Nas discussões considerem-se o Teorema 3.1 e uma associação (n,m,V).

Forte: n tem uma *definição de fato* de V, m tem um *uso de fato* de V, e todo *sub-caminho livre de definição* de n para m é um *sub-caminho livre de definição de fato* c.r.a V.

Neste caso, já que existe um *sub-caminho livre de definição de fato* c.r.a V de n para m; vai existir também uma *potencial associação* de n para m c.r.a. V.

Firme: n tem uma *definição de fato* de V, m tem um *uso de fato* de V, e pelo menos um sub-caminho de n para m é *livre de definição de fato* c.r.a. V, e pelo menos um sub-caminho de n para m é um sub-caminho *livre de definição possível* c.r.a V.

Como existe pelo menos um sub-caminho *livre de definição de fato* de n para m c.r.a V, existe uma potencial associação de n para m c.r.a V.

Fraca: n tem uma *definição de fato* de V, e nenhum sub-caminho de n para m é *livre de definição de fato* c.r.a V.

Para este tipo de associação podem ocorrer três casos diferentes para os Critérios Potenciais Usos. Considerando uma variável do tipo ponteiro p apontando para V (p=&V).

Caso 1 Nem a *definição* nem o *uso* é feito com dereferência de ponteiro ($*p$="valor"). Um sub-caminho *livre de definição possível* c.r.a V iria conter uma possível redefinição de V por um ponteiro dereferenciado. A abordagem Potencial Uso não identifica essa redefinição e considera o sub-caminho como livre de definição; desta forma, uma potencial associação definição uso é requerida.

Caso 2 O *uso* é feito por um ponteiro dereferenciado; já que não há redefinição explícita de V, a abordagem Potencial Uso requer a associação.

Caso 3 A *definição* é feita por um ponteiro dereferenciado. Existe uma definição intermediária da variável por um ponteiro dereferenciado. Essa definição não é considerada uma redefinição da variável devido ao Modelo de Fluxo de Dados dos Critérios Potenciais Usos; desta forma, a associação será requerida.

Muito Fraca: Ou a *definição* ou o *uso* ou ambos são *possíveis* ao invés de *de fato*.

Considerando que não há variáveis do tipo ponteiro indefinidas, a abordagem Potencial Uso vai sempre requerer pelo menos uma associação envolvida numa associação Muito Fraca.

Teorema 3.2 *Todos-Potenciais-Usos e Todos-Usos Muito Fraco são incomparáveis.*

Prova: Segundo a discussão acima; o Critério Todos-Potenciais-Usos não inclui Todos-Usos Muito Fraco, pois não requer todas as associações exigidas por esse critério quando o conjunto de sinônimos de um ponteiro tiver dois ou mais elementos. Por outro lado; o Critério Todos-Usos Muito Fraco não requer as associações para as quais não exista um uso explícito da variável, mesmo existindo um caminho livre de definição, portanto o Critério Todos-Usos Muito Fraco não inclui o Critério Todos-Potenciais-Usos. Assim Todos-Potenciais-Usos e Todos-Usos Muito Fraco são critérios incomparáveis.

Teorema 3.3 *Todos-Potenciais-Usos inclui Todos-Usos Horgan-London, Todos-Usos Forte, Todos-Usos Firme e Todos-Usos Fraco.*

Prova: Conforme discutido acima, toda associação de fluxo de dados definida pelos Critérios Todos-Usos Forte, Firme e Fraco são também exigidas pelo Critério Todos-Potenciais-Usos, portanto Todos-Potenciais-Usos inclui todos os outros. Todos-Potenciais-Usos inclui Todos-Usos de Horgan e London pois toda associação definição-uso, exigida pelo Critério Todos-Usos segundo a abordagem de Horgan e London, é também uma potencial associação de fluxo de dados – Teorema 3.4 – exigida pelo Critério Todos-Potenciais-Usos.

Como já foi dito, um dos aspectos essenciais do Teste Estrutural baseado em Análise de Fluxo de Dados é a identificação das associações de fluxo de dados; a ocorrência de ponteiros e outras estruturas de dados dificulta essa identificação. Esta seção apresenta uma abordagem conservadora para a identificação de associações de fluxo de dados que envolvem variáveis do tipo ponteiro; a abordagem conservadora pretende não deixar de garantir que

possíveis fluxos de dados dentro de um programa sejam exigidos, mesmo correndo o risco de exigir alguns que na verdade não existem. A abordagem apresentada foi comparada com outras duas abordagens que, de uma forma ou outra, buscam estender os critérios de fluxo de dados originalmente propostos por Rapps e Weyuker [RW85] para o teste de programas que usam ponteiros; a comparação entre as abordagens mostrou que a abordagem dos Critérios Potenciais Usos requer praticamente qualquer associação requerida pelas outras abordagens, além de algumas outras não exigidas pelas demais abordagens.

Ostrand e Weyuker afirmam que, além do problema de não identificar associações que realmente existem ou identificar associações que na verdade não existem, outro tipo de engano cometido por sistemas que analisam a semântica de ponteiros, aplicando uma análise de sinônimos muito conservadora, é que se pode gerar um número muito grande de associações definição-uso que na verdade nunca vão existir de fato na execução do programa.

Na verdade a visão demonstrada nessa argumentação é restrita e distorce o verdadeiro objetivo da atividade de teste, que é o de revelar a presença de defeitos no programa. Ao se estabelecer os elementos requeridos de um programa de uma maneira conservadora, mesmo correndo o risco de se estabelecer associações definição-uso que na verdade não existem, dá-se a chance de se revelar a presença de defeitos no programa. Nesse sentido a análise conservadora permite uma maior eficácia na atividade de teste.

3.2 Habilidade na Detecção de Defeitos

Esta seção apresenta uma comparação entre os Critérios Potenciais Usos, a Família de Critérios de Fluxo de Dados de Rapps e Weyuker e alguns outros critérios baseados em fluxo de dados e em fluxo de controle em termos de sua relativa habilidade de detectar a presença de defeitos em um programa.

Há algum tempo a relação de inclusão tem sido utilizada para comparar critérios de teste, como também para se definir ou selecionar um critério de teste. A relação de inclusão estabelece uma hierarquia (ordem parcial) entre critérios de teste, fornecendo informações sobre a força do teste. Infelizmente, o fato de um critério C_1 incluir um critério C_2 não traz muita informação sobre sua relativa habilidade de detectar defeitos; como o principal objetivo do teste é o de revelar a presença de um defeito ainda não descoberto, é natural buscar-se formas de se comparar critérios de teste segundo essa capacidade. A *habilidade de detectar defeitos* é outra forma de se comparar critérios de teste; o objetivo é encontrar alguma relação R entre os critérios C_1 e C_2 tal que se R(C_1,C_2) então pode-se garantir que C_1 é melhor para detectar defeitos que C_2.

Frankl e Weyuker [FW93b] propuseram um conjunto de relações mais fortes que a *relação de inclusão*. Foram definidas cinco novas relações: *restringe*, *cobre*, *particiona*, *cobre propriamente* e *particiona propriamente*; o foco está em como cada critério de teste divide o domínio de entrada em subdomínios. Elas analisaram essas relações sob o ponto de vista da habilidade de detecção de defeitos, de acordo com um conjunto de métricas probabilísticas. O conjunto de métricas de habilidade de detecção de defeitos está relacionado com a probabilidade que um conjunto de casos de teste tem de expor pelo menos um defeito. Essas relações e as métricas correspondentes são apresentadas na Seção 2.6.

A relação *cobre propriamente* foi escolhida para ser usada na construção da hierarquia entre os critérios. Frankl e Weyuker [FW93a] usaram essa relação para definir uma hierarquia entre a Família de Critérios de Fluxo de Dados de Rapps e Weyuker e alguns outros Critérios baseados em Análise de Fluxo de Dados. Se um critério C_1 cobre propriamente um critério

C_2, Frankl e Weyuker [FW93b] mostraram que $M_2(C_1) \geq M_2(C_2)$, ou seja, a probabilidade de que um conjunto de casos de teste C_1-adequado detecte a presença de pelo menos um defeito é maior que a de um conjunto de casos de teste C_2-adequado.

Nota-se que os resultado apresentados aqui são probabilísticos. Quando é mostrado que um critério C_1 cobre propriamente um critério C_2, mostra-se que conjuntos de casos de teste C_1-adequados são mais *predispostos* a detectar um defeito que conjuntos de casos de teste C_2-adequados. Desta forma resultados como C_1 cobre propriamente C_2 fornecem aos engenheiros de software uma importante razão para utilizar C_1 ao invés de utilizar C_2.

3.2.1 Definições

Uma *série de teste* ("test suite") é um multi-conjunto de casos de teste, cada um sendo composto de uma entrada ao programa e a respectiva saída esperada segundo a especificação. Muitas abordagens sistemáticas ao teste estão baseadas na idéia de se dividir o domínio de entrada do programa em sub-conjuntos, chamados *subdomínios*, e então requerer que séries de teste incluam elementos de cada subdomínio. Essas técnicas são muitas vezes referenciadas como *teste de partição* mas, na verdade, muitas delas subdividem o domínio de entrada em subdomínios sobrepostos, não formando uma verdadeira partição do domínio de entrada. Essas técnicas são mais propriamente denominadas como *teste baseado em subdomínio.*

No teste de subdomínio baseado no fluxo de controle do programa, cada subdomínio consiste de todos os elementos que levam um particular elemento de código a ser executado. Por exemplo, no teste de arcos cada subdomínio consiste em todas as entradas que causam a execução de um arco em particular; no teste de caminhos, cada subdomínio consiste em todas as entradas que causam a execução de um dado caminho. No teste baseado em análise de fluxo de dados usando o Critério Todos-Potenciais-Usos, cada subdomínio consiste de todas as entradas que executam um caminho que inclui um sub-caminho entre a definição de uma variável v e um ponto onde haja um potencial uso dessa variável. Na maioria dessas estratégias, muitos programas e especificações dão origem a subdomínios sobrepostos e duplicados.

Um critério de teste C é *baseado em subdomínio* se, para todo programa P e especificação S, existe um multi-conjunto não vazio $SD_C(P,S)$ de subdomínios (sub-conjuntos do domínio de entrada), tal que C requer a seleção de um ou mais casos de teste de cada subdomínio em $SD_C(P,S)$.

3.2.2 Hierarquia de Critérios

Nesta seção é definida uma hierarquia de critérios de teste baseada na relativa habilidade de detecção de defeitos. Inicialmente é apresentada uma hierarquia definida por Frankl e Weyuker [FW93a, FW93c] envolvendo os Critérios da Família de Critérios baseados em Análise de Fluxo de Dados, os Critérios K-Tuplas Requeridas definidos por Ntafos [Nta84] e os Critérios de Cobertura de Contexto definidos por Laski e Korel [LK83]. Em seguida são provadas algumas relações envolvendo a Família de Critérios Potenciais Usos [Mal91]. Mostra-se que os Critérios Potenciais Usos estabelecem uma hierarquia entre o teste de ramos ("Decision Coverage") e o teste de caminhos, mesmo quando a habilidade de detecção de defeitos é considerada.

Nesta seção, todos os critérios de teste discutidos são critérios baseados em sub-

domínio universalmente aplicáveis, isto é, os correspondentes critérios executáveis. No teste estrutural ou baseado em caminhos, a ocorrência de caminhos não executáveis no programa pode levar a requisitos de teste que não podem ser satisfeitos, representados pela ocorrência de subdomínios vazios em $SD_C(P,S)$.

A hierarquia apresentada por Frankl e Weyuker [FW93a, FW93c] é mostrada na Figura 3.4. Inicialmente Frankl e Weyuker provaram que os Critérios Todos-Usos ("All-Uses") e Todos-P-Usos ("All-P-Uses") cobrem propriamente uma variação do Critérios Todos-Arcos chamado *Cobertura de Decisões* ("Decision Coverage"). Na verdade, a variação está em como os arcos que deixam um comando de decisão são mapeados no caso de linguagens de programação em que a expressão de um comando de decisão pode não ser avaliada completamente antes de haver um fluxo de controle; por exemplo, na linguagem C, assim que se pode determinar com certeza que a expressão vai avaliar para "verdadeiro" ou para "falso" já ocorre o desvio do fluxo de controle, mesmo que a expressão não tenha sido avaliada por completo. Quando o critério de teste mapeia cada um dos possíveis desvios presentes em um comando de decisão separadamente ele é chamado de *Cobertura de Decisões.*

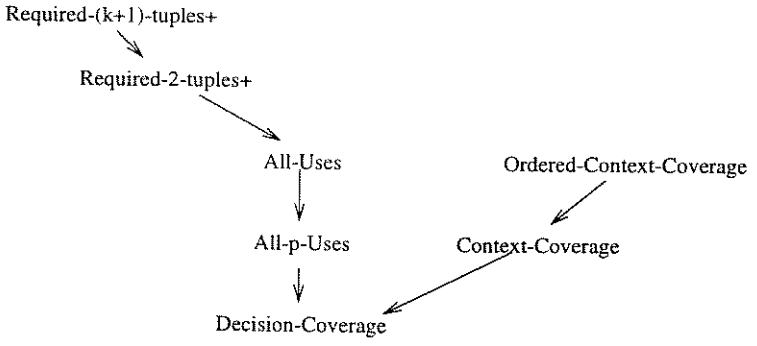

Figura 3.4: Relação Apresentada por Frankl e Weyuker

Em seguida são considerados os *Critérios K-Tuplas Requeridas* de Ntafos [Nta84]. Esses critérios requerem a execução de um sub-caminho a partir da definição de uma variável até um uso, influenciado por aquela definição, através de uma cadeia de definições e usos intermediários. Clarke et al. [CPRZ89] apontaram alguns problemas com a definição original de Ntafos e definiram os *Critérios K-Tuplas Requeridas+* com modificações para garantir que cada Critério K-Tuplas Requeridas inclua o Critério (K-1)-Tuplas Requeridas, onde K é o número de associações encadeadas. Frankl e Weyuker mostraram que para todo $K \geq 2$, o Critério (K+1)-Tuplas+ cobre propriamente o Critérios K-Tuplas+ e que o Critério 2-Tuplas+ cobre propriamente o Critérios Todos-Usos.

Por fim, Frankl e Weyuker consideram os critérios de Laski e Korel [LK83] conhecidos como *cobertura de contexto* ("context coverage") e *cobertura ordenada de contexto* ("ordered context coverage"). Esses critérios consideram caminhos passando por definições de todas as variáveis usadas em um determinado comando. Sejam $X_1, \cdots, X_k$ as variáveis usadas em um nó n. Um *contexto elementar de dados* ("elementary data context") para n é um conjunto $\{\delta_1, \cdots, \delta_k\}$ onde δ_i é uma definição de X_i e existe um sub-caminho p do nó de entrada ao nó n que inclui um caminho livre de definição c.r.a X_i de δ_i até n. Desta forma o fluxo de controle pode alcançar n com as variáveis $X_1, \cdots, X_k$ tendo os valores associados a elas nos nós $\delta_1, \cdots, \delta_k$, respectivamente. O Critério Cobertura de Contexto exige a execução desses sub-caminhos para cada contexto e o Critério Cobertura Ordenada de Contexto pede a execução dos sub-caminhos que percorrem os δ_i para todas as combinações possíveis considerando uma seqüência ordenada de definições. Frankl e Weyuker provam que o Critério Cobertura Ordenada de Contexto cobre propriamente o Critério Cobertura de

Contexto e que o Critério Cobertura de Contexto cobre propriamente o Critério Cobertura de Decisão. Prova-se também que nenhum desses dois critérios cobre propriamente o Critério Todos-Usos ou o Critério Todos-P-Usos.

Frankl e Weyuker [FW93c] também consideraram a Análise de Mutantes [DLS78] na definição de sua hierarquia de critérios baseada na habilidade de detecção de defeitos. Ao contrário dos outros critérios analisados, a Análise de Mutantes não é orientada a caminhos. Ao invés disso, ela considera um conjunto de casos de teste T como sendo adequado para testar um programa P se T distingue P de uma conjunto de variações (programas), não equivalentes, de P chamados *mutantes*. Esses mutantes são criados através da aplicação de *operadores de mutação*. Cada operador de mutação é uma regra para se aplicar uma transformação sintática simples em P. Na análise de mutantes, os subdomínios são da forma $\{t \mid P(t) \neq P'(t)\}$ onde P' é um mutante de P. Frankl e Weyuker definiram uma variação da análise de mutantes que inclui apenas dois operadores: i) substituir uma decisão por *verdadeiro*; e, ii) substituir uma decisão por *falso* – essa variação foi denominada *mutação limitada* ("limited mutation"). Segundo Budd [Bud81] a inclusão desses dois operadores garante que a análise de mutantes inclui o teste de ramos.

Frankl e Weyuker mostraram então que a mutação limitada não cobre o teste de decisões. Essa conclusão apesar de relevante não é significativa no contexto desta tese. Investigações mais cuidadosas devem ser conduzidas no sentido de definir o relacionamento entre a análise de mutantes e critérios estruturais, tanto sob o ponto de vista teórico quanto empírico. Grande número de trabalhos tem surgido nos últimos anos relacionados a esse tema [ORZ93, MW93, MW94a, MW94b, MW95].

A hierarquia apresentada na Figura 3.4, representando as provas apresentadas por Frankl e Weyuker, revelam uma deficiência desses critérios no que diz respeito a um dos requisitos mínimos de uma Família de Critérios de Teste, o de estabelecer uma ordem parcial entre a cobertura de ramos e o teste de caminhos. A seguir são apresentadas algumas discussões envolvendo os Critérios Potenciais Usos com o objetivo de mostrar que essa Família de Critérios estabelece uma ordem parcial entre o teste de ramos e o teste de caminhos mesmo quando os aspectos relacionados à habilidade na detecção de defeitos são considerados.

Os teoremas 3.4 e 3.5, provados por Maldonado [Mal91], apresentam algumas características relacionadas à definição dos Critérios Potenciais Usos e são usados para provar que o Critério Todos-Potenciais-Usos cobre propriamente o Critério Todos-Usos e que o Critério Todos-Potenciais-DU-Caminhos cobre propriamente o Critério Todos-DU-Caminhos.

Teorema 3.4 (Potencial Associação Definição-Uso) *Toda* associação definição-uso (m, n, V) *de um módulo* M *é também uma* potencial associação definição-uso.

Teorema 3.5 (Potencial DU-Caminhos) *Todo DU-Caminho é também um* Potencial DU-Caminho

Frankl e Weyuker [FW93b] provam que se $SD_{C_2}(P,S) \subseteq SD_{C_1}(P,S)$; ou seja, se o multi-conjunto de subdomínios definidos pelo Critério C_2 for um submulti-conjunto do multi-conjunto de subdomínios do Critério C_1, então C_1 cobre, particiona, cobre propriamente e particiona propriamente C_2 para (P,S).

Observação 1: $SD_{Todos-Usos}(P,S) \subseteq SD_{Todos-Potenciais-Usos}(P,S)$. O critério *Todos-Usos* requer que associações definição-uso sejam cobertas pelos casos de teste. Cada subdomínio $D_i \in SD_{Todos-Usos}(P,S)$ corresponde ao conjunto

de elementos no domínio de entrada que executa a associação i. Como cada associação definição-uso também é uma potencial associação definição-uso requerida por *Todos-Potenciais-Usos* (Teorema 3.4), cada $D_i \in SD_{Todos-Usos}(P,S)$ é também um elemento de $SD_{Todos-Potenciais-Usos}(P,S)$; conseqüentemente, $SD_{Todos-Usos}(P,S) \subseteq SD_{Todos-Potenciais-Usos}(P,S)$.

Teorema 3.6 Todos-Potenciais-Usos $\overset{\text{cobre propriamente}}{\Longrightarrow}$ Todos-Usos.

Prova:

Segue diretamente do fato de que $SD_{Todos-Usos}(P,S) \subseteq SD_{Todos-Potenciais-Usos}(P,S)$, *Observação 1.*

Teorema 3.7 Todos-Usos $\overset{\text{cobre propriamente}}{\not\Longrightarrow}$ Todos-Potenciais-Usos.

Prova:

Deve-se mostrar que existe um subdomínio definido por *Todos-Potenciais-Usos* que não pode ser representado por uma união de subdomínio definidos por *Todos-Usos*. Considere o exemplo na Figura 3.5.

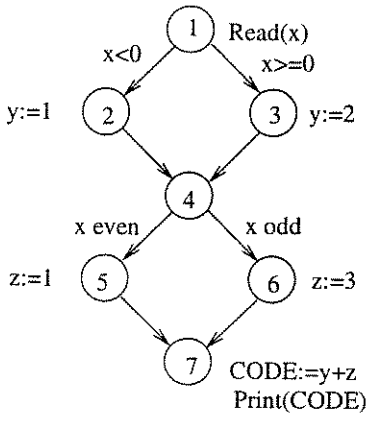

Figura 3.5: Exemplo - Relação Não Cobre.

O conjunto de associações requeridas por Todos-Usos é: [{1,(1,2),x}, {1,(1,3),x}, {1,(4,5),x}, {1,(4,6),x}, {2,7,y}, {3,7,y}, {5,7,z}, {6,7,z}]; esse conjunto de associações divide o domínio de entrada em quatro subdomínios: $\{x < 0,\ x \geq 0,\ x = even,\ x = odd\}$. Além desse conjunto de associações, o critério Todos-Potenciais-Usos também requer {2,5,y} e {2,6,y}; conseqüentemente, dois subdomínios seriam definidos $D_1 = \{x/x < 0\ e\ x = even\}$ e $D_2 = \{x/x \geq 0\ e\ x = odd\}$. Não existe nenhuma união de subdomínios em Todos-Usos que seja equivalente a D_1 ou D_2, portanto *Todos-Usos* $\overset{\text{cobre}}{\not\Longrightarrow}$ *Todos-Potenciais-Usos* e, conseqüentemente, *Todos-Usos* $\overset{\text{cobre propriamente}}{\not\Longrightarrow}$ *Todos-Potenciais-Usos.*

Observação 2: $SD_{Todos-DU-Caminhos}(P,S) \subseteq SD_{Todos-Potenciais-DU-Caminhos}(P,S)$. A prova é análoga à da Observação 1, desta vez considerando o Teorema 3.5.

Teorema 3.8 Todos-Potenciais-DU-Caminhos $\overset{\text{cobre propriamente}}{\Longrightarrow}$ Todos-DU-Caminhos.

Prova: Segue imediatamente do fato de que $SD_{Todos-DU-Caminhos}(P,S) \subseteq SD_{Todos-Potenciais-DU-Caminhos}(P,S)$, *Observação 2.*

Teorema 3.9 Todos-DU-Caminhos $\overset{\text{cobre propriamente}}{\not\Longrightarrow}$ Todos-Potenciais-DU-Caminhos.

Prova: É análoga à prova do Teorema 3.7.

Prova-se agora que o Critério Todos-Potenciais-DU-Caminhos cobre propriamente o Critério Todos-Potenciais-Usos/DU.

Teorema 3.10 Todos-Potenciais-DU-Caminhos $\overset{\text{cobre propriamente}}{\Longrightarrow}$ Todos-Potenciais-Usos/DU.

Prova: Considere uma potencial associação (i,j,x), o conjunto de *Potenciais DU-Caminhos* que cobrem essa associação $p_1,\cdots,p_k$ e o conjunto de subdomínios definidos por esses caminhos $d_1,\cdots,d_k$. O critério *Todos-Potenciais-DU-Caminhos* requer a execução de todo *Potencial DU-Caminho* cobrindo uma associação; desta forma, existem k subdomínios correspondendo à associação (i,j,x): $d_1,\cdots,d_k$. O critério *Todos-Potenciais-Usos/DU* considera a associação coberta se qualquer *Potencial DU-Caminho* é executado; desta forma o subdomínio correspondendo à associação (i,j,x) é $d = d_1 \cup \cdots \cup d_k$. Como isso é válido para toda potencial associação, para todo subdomínio $D \in SD_{Todos-Potenciais-Usos/DU}$ existe um conjunto de subdomínios $D_1,\cdots,D_k \in SD_{Todos-Potenciais-DU-Caminhos}$ tal que $D = D_1 \cup \cdots \cup D_k$; portanto o Critério Todos-Potenciais-DU-Caminhos cobre propriamente o Critério Todos-Potenciais-Usos/DU.

Para completar a hierarquia mostrada na Figura 3.6 basta mostrar que o Critério Todos-Potenciais-Usos/DU não cobre propriamente o Critério Todos-Potenciais-Usos.

Teorema 3.11 Todos-Potenciais-Usos/DU $\overset{\text{cobre propriamente}}{\not\Longrightarrow}$ Todos-Potenciais-Usos.

Prova: O Critério Todos-Potenciais-Usos exige que se executem caminhos que percorram sub-caminhos cobrindo cada uma das potenciais associações de fluxo de dados definidas em um programa. Não importa se o sub-caminho percorrido é ou não um Potencial DU-Caminho. Assim, cada potencial associação do programa dá origem a um subdomínio D do domínio de entrada, consistindo de todos os elementos que causam a execução de um caminho que inclua um sub-caminho que percorre a associação. O subdomínio D está dividido em duas partes $D = D_p \cup D_n$, onde D_p contém os elementos que cobrem a associação percorrendo um Potencial DU-Caminho, e D_n são os que cobrem a associação percorrendo um sub-caminho que não é um Potencial DU-Caminho,

ou seja, esses sub-caminhos incluem ciclos. Por outro lado, o Critério Todos-Potenciais-Usos/DU exige que a associação seja coberta por um Potencial DU-Caminho, definindo para cada potencial associação um subdomínio que inclui apenas os elementos em D_p. Desta forma não há como formar os subdomínios definidos por Todos-Potenciais-Usos a partir da união de subdomínios definidos por Todos-Potenciais-Usos/DU; portanto Todos-Potenciais-Usos/DU não cobre propriamente Todos-Potenciais-Usos.

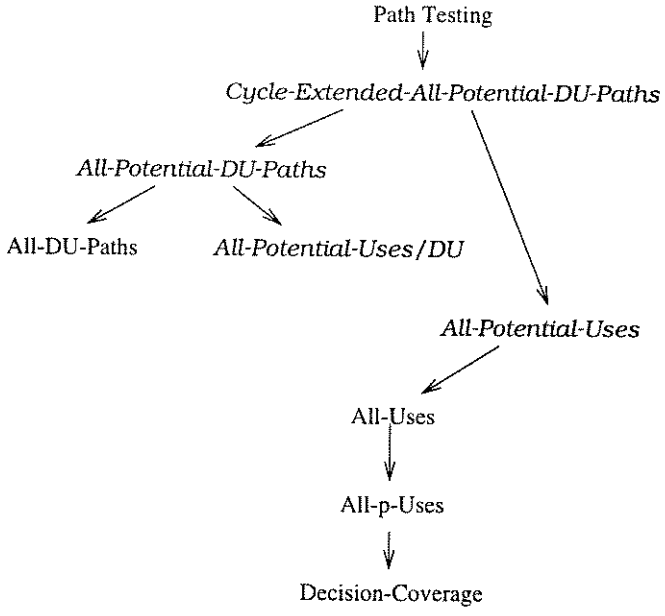

Figura 3.6: Relação Cobre Propriamente

Observação 3: qualquer sub-caminho π que cubra uma potencial associação de fluxo de dados c.r.a x pertence à extensão a ciclo de algum Potencial DU-Caminho c.r.a x. *Prova*: Se π for um Potencial DU-Caminho, por definição ele pertence à extensão-a-ciclo(π, x). Caso contrário π contém um ou mais ciclos; neste caso, $\pi \in$ extensão-a-ciclo(π', x) onde π' é o Potencial DU-Caminho obtido eliminando-se os ciclos de π.

Como conseqüência desta observação, qualquer elemento do domínio de entrada que exercite uma associação fará parte do domínio definido pela extensão a ciclo de algum Potencial DU-Caminho que exercite a associação.

Teorema 3.12 Todos-Potenciais-DU-Caminhos Estendido a Ciclo $\overset{\text{cobre propriamente}}{\Longrightarrow}$ Todos-Potenciais-Usos.

Prova: Dada uma potencial associação de fluxo de dados c.r.a x e um subdomínio D dos elementos do domínio de entrada que percorrem essa associação, D é o subdomínio definido pelo Critério Todos-Potenciais-Usos. Qualquer Potencial-DU-Caminho π que percorra a potencial associação define um subdomínio $D_\pi \subseteq D$. Cada sub-caminho $\pi' \in$ extensão-a-ciclo(π, x) também

percorre a associação, assim o subdomínio $D_{\pi'} \subseteq D$. Considerando o conjunto de Potenciais DU-Caminhos que percorrem a associação como sendo $(\pi_1, \pi_2, \cdots, \pi_n)$ e o conjunto de extensões a ciclo como sendo, para cada π_i, $(\pi_i^1, \pi_i^2, \cdots, \pi_i^k)$ representados por Π_i', segundo a definição acima ter-se-ia $D_{\Pi_1'} \cup D_{\Pi_2'} \cup \cdots \cup D_{\Pi_n'} = D$; ou seja, o subdomínio definido por Todos-Potenciais-Usos é formado pela união dos subdomínios definidos por Todos-Potenciais-DU-Caminhos Estendido a Ciclo. Portanto Todos-Potenciais-DU-Caminhos Estendido a Ciclo cobre propriamente Todos-Potenciais-Usos

Teorema 3.13 Todos-Potenciais-DU-Caminhos Estendido a Ciclo $\overset{\text{cobre propriamente}}{\Longrightarrow}$ Todos-Potenciais-DU-Caminhos.

Prova: Por definição tem-se que o multi-conjunto de subdomínios definido por Todos-Potenciais-DU-Caminhos é um submulti-conjunto do multi-conjunto de subdomínios definido por Todos-Potenciais-DU-Caminhos Estendido a Ciclo; assim, segundo Frankl e Weyuker [FW93b], o Critério Todos-Potenciais-DU-Caminhos Estendido a Ciclo cobre propriamente o Critério Todos-Potenciais-DU-Caminhos.

Teorema 3.14 Todos-Caminhos $\overset{\text{cobre propriamente}}{\Longrightarrow}$ Todos-Potenciais-DU-Caminhos Estendido a Ciclo.

Prova: Como todo e qualquer sub-caminho requerido pelo Critério Todos-Potenciais-DU-Caminhos Estendido a Ciclos é parte de um ou mais caminhos requeridos pelo Critério Todos-Caminhos, o multi-conjunto de subdomínios definidos pelo Critério Todos-Potenciais-DU-Caminhos Estendido a Ciclos é um submulti-conjunto do multi-conjunto de subdomínios definidos pelo Critério Todos-Caminhos assim o Critério Todos-Caminhos cobre propriamente o Todos-Potenciais-DU-Caminhos Estendido a Ciclos.

Desta forma prova-se que uma hierarquia de critérios entre a cobertura de ramos e o teste de caminhos é estabelecida pelos Critérios Potenciais Usos, mesmo considerando a habilidade de detecção de defeitos. Essa hierarquia de critérios é mostrada na Figura 3.6.

3.3 Considerações Finais

Este Capítulo apresentou dois estudos relacionados aos Critérios Potenciais Usos no nível do teste de unidade; um estudo sobre a aplicação desses critérios em programas com ponteiros e a análise da habilidade de detectar defeitos. Esses estudos complementam os apresentados por Maldonado [Mal91], no sentido de definir e analisar as propriedades dos Critérios Potenciais Usos.

A comparação entre os Critérios Potenciais Usos e os critérios da Família de Critérios de Fluxo de Dados segundo sua relativa habilidade de detecção de defeitos é fundamental para a aplicação prática desses critérios; com ela pode-se decidir, de acordo com os recursos disponíveis, quais critérios utilizar para se testar determinado programa. A comparação

apresentada envolve os correspondentes critérios executáveis. É importante notar que o relacionamento entre os critérios executáveis pode ser diferente do relacionamento entre os critérios originais. No entanto, para critérios baseados no mesmo tipo de elemento requerido, por exemplo, associações de fluxo de dados, arcos, etc., a hierarquia com relação à habilidade de detecção de defeitos permanece a mesma quando os critérios básicos são considerados. Essa propriedade pode ser demonstrada a partir do fato de que ambos os critérios geram, para cada elemento requerido não executável, um subdomínio vazio em $SD_C(P, S)$; como o mesmo número de subdomínios vazios são gerados para os dois critérios, eles não alteram as relações da hierarquia.

Capítulo 4

Extensão para o Teste de Integração

Neste capítulo definem-se a Família de Critérios Potenciais Usos para o Teste de Integração e a correspondente Família de Critérios Potenciais Usos para o Teste de Integração Executável. A terminologia apresentada na Seção 2.1 é utilizada na definição dos *Critérios Potenciais Usos de Integração*; facilitando a comparação entre esses critérios e os outros critérios para o teste de integração apresentados na Seção 2.5. Essas comparações encontram-se no final deste capítulo.

4.1 Critérios Potenciais Usos para o Teste de Integração

Vários Critérios de Teste Estrutural Baseados em Análise de Fluxo de Dados têm sido definidos [Her76, LK83, Nta84, RW85, UY88, Mal91]. Como regra geral, esses critérios consideram informações restritas ao escopo de um único módulo do programa para derivar requisitos de teste. A partir do início da década de 90 tem surgido a preocupação com a definição de critérios de teste específicos para o teste de integração [HS91, LM90, LW90]. A maioria das abordagens propostas para o Teste de Integração estão baseadas na Família de Critérios proposta por Rapps e Weyuker [RW85] para o teste de unidade; conseqüentemente, requerem a ocorrência de um uso explícito de variável para estabelecer uma associação de fluxo de dados.

A Família de Critérios definida neste capítulo está baseada na Família de Critérios Potenciais Usos; estão portanto, apoiados no conceito de *potencial uso*. Assim como na abordagem de Linnenkugel e Müllerburg [LM90], o modelo de integração utilizado no escopo dos Critérios Potenciais Usos de Integração representa o programa por um grafo de chamada. Como descrito no Capítulo 2, um programa é representado por um multi-grafo direcionado, onde módulos são associados a nós e os possíveis fluxos de controle entre módulos são representados por arcos. O grafo de chamada é um multi-grafo; podem haver mais de uma chamada entre dois módulos, o que corresponde a mais de um arco ligando os respectivos nós do grafo. Por *módulo* entende-se uma sub-rotina, sub-programa ou função, conceito presente na definição de linguagens procedimentais; esse conceito representa a idéia de uma parte separada do código do programa sobre a qual incidam regras específicas de ativação ou chamada.

A abordagem considera a integração dos módulos feita *dois-a-dois* ("pairwise"); os requisitos de teste são, conseqüentemente, derivados para cada par de módulos. A abordagem dois-a-dois apóia as estratégias incrementais de integração: "Bottom-Up", "Top-Down" e "Sanduíche", desde que cada incremento seja aplicado a um par de módulos de cada vez.

Definição 4.1 *Uma Potencial Associação Definição-Uso Interprocedimental de Chamada é uma quádrupla* (i, j, x, c) *onde* c *é uma chamada de um módulo* M_1 *para um módulo* M_2*;* $x \in in_c$*;* i *é um nó de* M_1 *que contém uma definição global de* x *e existe um sub-caminho* $(i) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (j)$ *tal que* p *é livre de definição c.r.a* x *e* $q \cdot (j)$ *é livre de definição c.r.a* $formal_c(x)$*.*

Definição 4.2 *Uma Potencial Associação Definição-Uso Interprocedimental de Retorno é uma quádrupla* (i, j, x, c) *onde* c *é uma chamada de um módulo* M_1 *para um módulo* M_2*;* $x \in out_c$*;* i *é um nó de* M_2 *que contém uma definição global de* $formal_c(x)$ *e existe um sub-caminho* $(i) \cdot p \cdot (n_{out}(M_2)) \cdot (n_c) \cdot q \cdot (j)$ *tal que* p *é livre de definição c.r.a* $formal_c(x)$ *e* $q \cdot (j)$ *é livre de definição c.r.a* x*.*

Definição 4.3 *Um sub-caminho* $(n_i) \cdot q \cdot (n_c) \cdot (n_{in}(M_2)) \cdot p \cdot (n_j, n_k)$*; onde* n_c *é o nó da chamada* c *de* M_1 *para* M_2*,* $n_i \in M_1$ *e* $(n_j, n_k) \in M_2$*, é um* Potencial DU-Caminho Interprocedimental de Chamada *c.r.a* x*, se e somente se* $(n_i) \cdot q \cdot (n_c)$ *é um Potencial DU-Caminho c.r.a* x *e* $(n_{in}(M_2)) \cdot p \cdot (n_j, n_k)$ *é um Potencial DU-Caminho c.r.a* $formal_c(x)$*.*

Definição 4.4 *Um sub-caminho* $(n_i) \cdot q \cdot (n_{out}(M_2)) \cdot (n_c) \cdot p \cdot (n_j, n_k)$*; onde* n_c *é o nó da chamada* c *de* M_1 *para* M_2*,* $n_i \in M_2$ *e* $(n_j, n_k) \in M_1$*, é um* Potencial DU-Caminho Interprocedimental de Retorno *c.r.a* $formal_c(x)$*, se e somente se* $(n_i) \cdot q \cdot (n_{out}(M_2))$ *é um Potencial DU-Caminho c.r.a* $formal_c(x)$ *e* $(n_c) \cdot p \cdot (n_j, n_k)$ *é um Potencial DU-Caminho c.r.a* x*.*

Uma *Potencial Associação de Fluxo de Dados Interprocedimental* é uma Potencial Associação Definição Uso Interprocedimental de Chamada ou de Retorno. Um *Potencial DU-Caminho Interprocedimental* é um Potencial DU-Caminho Interprocedimental de Chamada ou de Retorno.

Segue a definição da Família de Critérios Potenciais Usos de Integração básicos:

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-Potenciais-Usos* (*INT-All-Potential-Uses*), se e somente se, para cada chamada c do módulo M_1 para o módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ para todo nó n com um potencial uso de $formal_c(x)$ definido em $n_{in}(M_2)$, com p livre de definição c.r.a x e q livre de definição c.r.a $formal_c(x)$, e pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ para todo arco (n, n') com um potencial uso de $formal_c(x)$ definido em $n_{in}(M_2)$, com p livre de definição c.r.a x e $q \cdot (n)$ livre de definição c.r.a $formal_c(x)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ para todo nó m com um potencial uso de x definido em n_c, com p livre de definição c.r.a $formal_c(x)$ e q livre de definição c.r.a x, e pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ para todo arco (m, m') com um potencial uso de x definido em n_c, com p livre de definição c.r.a $formal_c(x)$ e $q \cdot (m)$ livre de definição c.r.a x.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-Potenciais-Usos/DU* (*INT-All-Potential-Uses/DU*), se e somente se para, cada chamada c do módulo M_1 ao módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ onde n tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x e $(n_{in}(M_2)) \cdot q \cdot (n)$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ e pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ onde (n, n') tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x e $(n_{in}(M_2)) \cdot q \cdot (n, n')$ é um Potencial DU-Caminho c.r.a $formal_c(x)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ onde m tem um potencial uso de x e $(n) \cdot p \cdot (n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ e $(n_c) \cdot q \cdot (m)$ é um Potencial DU-Caminho c.r.a x e pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ onde (m, m') tem um potencial uso de x e $(n) \cdot p \cdot (n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ e $(n_c) \cdot q \cdot (m, m')$ é um Potencial DU-Caminho c.r.a x.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-Potenciais-DU-Caminhos* (*INT-All-Potential-DU-Paths*), se e somente se, para cada chamada c do módulo M_1 ao módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir todo sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ onde n tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x e $(n_{in}(M_2)) \cdot q \cdot (n)$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ e todo sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ onde (n, n') tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x e $(n_{in}(M_2)) \cdot q \cdot (n, n')$ é um Potencial DU-Caminho c.r.a $formal_c(x)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir todo sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ onde m tem um potencial uso de x e $(n) \cdot p \cdot (n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ e $(n_c) \cdot q \cdot (m)$ é um Potencial DU-Caminho c.r.a x e todo sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ onde (m, m') tem um potencial uso de x e $(n) \cdot p \cdot (n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ e $(n_c) \cdot q \cdot (m, m')$ é um Potencial DU-Caminho c.r.a x.

Uma das propriedades que devem ser satisfeitas por um bom critério de teste é a aplicabilidade [FW86]; 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 seja C-adequado para P; ou seja, o conjunto de caminhos executados por T inclui cada elemento exigido pelo critério C.

Tanto os Critérios Potenciais Usos para o teste de unidade como os outros critérios de teste estrutural apresentados no Capítulo 2 não satisfazem a propriedade de aplicabilidade. Esses critérios foram modificados no intuito de satisfazer essa propriedade, dando origem aos chamados *critérios executáveis*. Os Critérios Potenciais Usos Executáveis para o Teste de Unidade foram definidos por Maldonado [Mal91].

Em geral é indecidível se um dado caminho ou associação é ou não executável; conseqüentemente, é indecidível se existe um conjunto de casos de teste T que seja C-adequado a um dado programa P.

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 [Fra87]. Mesmo porque, sabe-se que a presença de caminhos não executáveis modifica consideravelmente algumas das propriedades dos critérios; por exemplo, Frankl [Fra87, FW88] observou que a relação de inclusão entre os Critérios da Família de Critérios de Fluxo de Dados muda significativamente.

Define-se o conjunto $E_a(M_1, M_2)$ como sendo o conjunto de associações de fluxo de dados interprocedimentais executáveis presentes na conexão do módulo M_1 com o módulo M_2. E $E_c(M_1, M_2)$ como sendo o conjunto de *Potenciais DU-Caminhos Interprocedimentais* executáveis presentes na conexão do módulo M_1 com o módulo M_2.

Na modificação introduzida nos Critérios Potenciais Usos de Integração, para definir os correspondentes critérios executáveis, considera-se que uma potencial associação ou um potencial-du-caminho interprocedimental só será requerido pelo critério se pertencerem, respectivamente, a $E_a(M_1, M_2)$ ou a $E_c(M_1, M_2)$. Assim, são definidos os critérios *INT-Todos-Potenciais-Usos Executáveis*, *INT-Todos-Potenciais-Usos/DU Executáveis* e *INT-Todos-Potenciais-DU-Caminhos Executáveis*, representados por *INT-Todos-Potenciais-Usos**, *INT-Todos-Potenciais-Usos/DU** e *INT-Todos-Potenciais-DU-Caminhos**.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-Potenciais-Usos** (*INT-All-Potential-Uses**), se e somente se, para cada chamada c do módulo M_1 ao módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ para todo nó n com um potencial uso de $formal_c(x)$ definido em $n_{in}(M_2)$, com p livre de definição c.r.a x e q livre de definição c.r.a $formal_c(x)$, e pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ para todo arco (n, n') com um potencial uso de $formal_c(x)$ definido em $n_{in}(M_2)$, com p livre de definição c.r.a x e $q \cdot (n)$ livre de definição c.r.a $formal_c(x)$; com as potenciais associações entre (m) e (n) por n_c e entre (m) e (n, n') por n_c pertencendo à $E_a(M_1, M_2)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ para todo nó m com um potencial uso de x definido em n_c, com p livre de definição c.r.a $formal_c(x)$ e q livre de definição c.r.a x, e pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ para todo arco (m, m') com um potencial

uso de x definido em n_c, com p livre de definição c.r.a $formal_c(x)$ e $q \cdot (m)$ livre de definição c.r.a x; com as potenciais associações entre (n) e (m) por $n_{out}(M_2)$ e entre (n) e (m, m') por $n_{out}(M_2)$ pertencendo à $E_a(M_1, M_2)$.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-Potenciais-Usos/DU** (*INT-All-Potential-Uses/DU**), se e somente se, para cada chamada c do módulo M_1 ao módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ onde n tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$ e $(n_{in}(M_2)) \cdot q \cdot (n)$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$ e pelo menos um sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ onde (n, n') tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$ e $(n_{in}(M_2)) \cdot q \cdot (n, n')$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ onde m tem um potencial uso de x e $(n) \cdot p \cdot (n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$ e $(n_c) \cdot q \cdot (m)$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$ e pelo menos um sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ onde (m, m') tem um potencial uso de x e $(n) \cdot p \cdot (n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$ e $(n_c) \cdot q \cdot (m, m')$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$.

A tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-Potenciais-DU-Caminhos** (*INT-All-Potential-DU-Paths**), se e somente se, para cada chamada c do módulo M_1 ao módulo M_2:

- para toda definição $d_m(M_1, x)$ com $x \in in_c$ que alcança n_c por um sub-caminho livre de definição c.r.a x, Φ incluir todo sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ onde n tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$ e $(n_{in}(M_2)) \cdot q \cdot (n)$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$ e todo sub-caminho $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n, n')$ onde (n, n') tem um potencial uso de $formal_c(x)$ e $(m) \cdot p \cdot (n_c)$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$ e $(n_{in}(M_2)) \cdot q \cdot (n, n')$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$; e,
- para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$ que alcança $n_{out}(M_2)$ por um sub-caminho livre de definição c.r.a $formal_c(x)$, Φ incluir todo sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ onde m tem um potencial uso de x e $(n) \cdot p \cdot (n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$ e $(n_c) \cdot q \cdot (m)$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$ e todo sub-caminho $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$ onde (m, m')

tem um potencial uso de x e $(n)\cdot p\cdot(n_{out}(M_2))$ é um Potencial DU-Caminho c.r.a $formal_c(x)$ em $E_c(M_1, M_2)$ e $(n_c)\cdot q\cdot(m, m')$ é um Potencial DU-Caminho c.r.a x em $E_c(M_1, M_2)$.

Com a definição desses critérios deixa-se de ter o problema da indecidibilidade sobre a existência de um conjunto de casos de teste T que seja C-adequado para um dado programa P, mas passa-se a ter o problema de reconhecer ou avaliar se um dado conjunto de casos de teste T é C^*-adequado para um dado programa P. Esse reconhecimento é geralmente feito de maneira não automática, através da identificação de requisitos de teste não executáveis.

O Critério Todos-Potenciais-DU-Caminhos foi estendido utilizando-se o conceito de extensão a ciclo [FW86] com o objetivo principal de obter-se uma hierarquia de critérios que incluísse os critérios Todos-Arcos e Todas-Definições, mesmo na presença de caminhos não executáveis. Da mesma forma o Critério INT-Todos-Potenciais-DU-Caminhos pode ser estendido usando-se o conceito de extensão a ciclo, dando origem ao *Critério INT-Todos--Potenciais-DU-Caminhos-Estendido-a-Ciclo* ("INT-Cycle-Extended-Potential-DU-Paths"). De forma análoga pode se definir o correspondente critério estendido a ciclo executável, dando origem ao *Critério INT-Todos-Potenciais-DU-Caminhos-Estendido-a-Ciclo**.

4.2 Exemplos

Nesta seção são apresentados dois exemplos utilizando os Critérios Potenciais Usos de Integração. O primeiro exemplo ilustra a diferença entre cada um dos critérios da Família de Critérios Potenciais Usos de Integração Básicos e o segundo exemplo compara os Critérios Potenciais Usos de Integração com a estratégia para o teste de integração proposta por Harrold e Soffa [HS91].

Considerando-se o grafo de programa para os módulos A e B da Figura 4.1 onde o módulo A chama o módulo B no nó 7, segue o conjunto de associações e o conjunto de sub-caminhos exigidos pelos Critérios Potenciais Usos de Integração.

Para os Critérios INT-Todos-Potenciais-Usos e INT-Todos-Potenciais-Usos/DU são estabelecidas as seguintes associações:

Considere a seguinte notação $< n, (j,k), \{v_1, v_2, \cdots, v_m\} >$ representando a associação do nó n para o nó k ou arco (j,k) com relação às variáveis $\{v_1, v_2, \cdots, v_m\}$. A notação utilizada considera o conceito de arco essencial [Chu87] para representar as associação, o nó que faz parte da associação está implícito no arco essencial utilizado; o conceito de arco essencial utilizado neste contexto reduz o número de associações requeridas, eliminando as redundantes.

Do módulo A para o módulo B:

```
INT-Todos-Potenciais-Usos e INT-Todos-Potenciais-Usos/DU.
A --> B

1- <1,(2,7),{x,y}> <1,(1,2),{m,n}>
2- <1,(2,7),{x,y}> <1,(1,3),{m,n}>
3- <1,(2,7),{x,y}> <1,(4,5),{m,n}>
4- <1,(2,7),{x,y}> <1,(4,6),{m,n}>
```

Do módulo B para o módulo A:

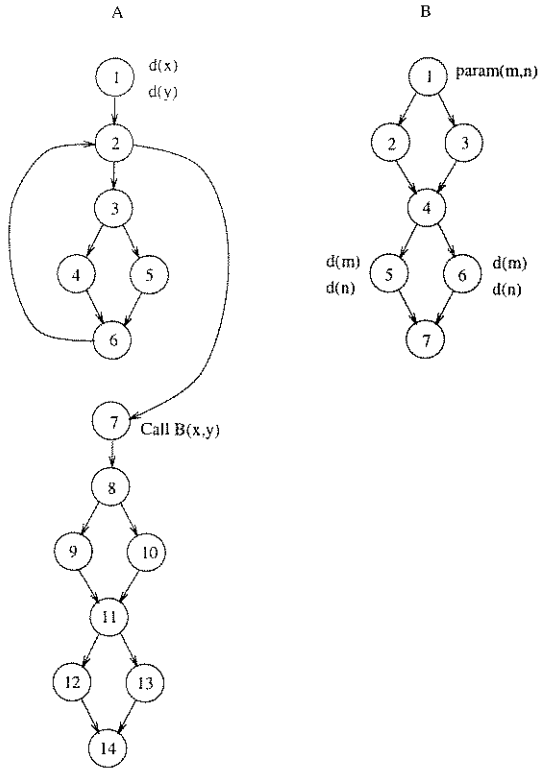

Figura 4.1: Exemplo para os Critérios Potenciais Usos de Integração.

```
INT-Todos-Potenciais-Usos e INT-Todos-Potenciais-Usos/DU.
B --> A

5-  <5,(5,7),{m,n}> <7,(8,9),{x,y}>
6-  <5,(5,7),{m,n}> <7,(8,10),{x,y}>
7-  <5,(5,7),{m,n}> <7,(11,12),{x,y}>
8-  <5,(5,7),{m,n}> <7,(11,13),{x,y}>
9-  <6,(6,7),{m,n}> <7,(8,9),{x,y}>
10- <6,(6,7),{m,n}> <7,(8,10),{x,y}>
11- <6,(6,7),{m,n}> <7,(11,12),{x,y}>
12- <6,(6,7),{m,n}> <7,(11,13),{x,y}>
```

Para o Critério INT-Todos-Potenciais-DU-Caminhos são estabelecidos os seguintes sub-caminhos:

Do módulo A para o módulo B:

```
INT-Todos-Potenciais-DU-Caminho.
A --> B

1- A1 A2 A7    B1 B2 B4 B5
2- A1 A2 A7    B1 B2 B4 B6
3- A1 A2 A7    B1 B3 B4 B5
4- A1 A2 A7    B1 B3 B4 B6
```

Do módulo B para o módulo A:

```
INT-Todos-Potenciais-DU-Caminho.
B --> A

5-  B5 B7    A8 A9 A11 A12 A14
6-  B5 B7    A8 A9 A11 A13 A14
7-  B5 B7    A8 A10 A11 A12 A14
8-  B5 B7    A8 A10 A11 A13 A14
9-  B6 B7    A8 A9 A11 A12 A14
10- B6 B7    A8 A9 A11 A13 A14
11- B6 B7    A8 A10 A11 A12 A14
12- B6 B7    A8 A10 A11 A13 A14
```

Considerando-se o conjunto de elementos requeridos apresentados acima, seguem exemplos de conjuntos de caminhos que satisfazem cada um dos critérios.

O seguinte conjunto de caminhos satisfaz o Critérios INT-Todos-Potenciais-Usos:

```
1-  A1 A2 A3 A4 A5 A2 A7 B1 B2 B4 B5 B7 A8 A9 A11 A12 A14
2-  A1 A2 A3 A4 A5 A2 A7 B1 B3 B4 B6 B7 A8 A9 A11 A13 A14
3-  A1 A2 A3 A4 A5 A2 A7 B1 B2 B4 B6 B7 A8 A10 A11 A12 A14
4-  A1 A2 A3 A4 A5 A2 A7 B1 B2 B4 B5 B7 A8 A10 A11 A13 A14
```

Esse conjunto não satisfaz o Critério INT-Todos-Potenciais-Usos/DU, já que a seqüência inicial de nós não forma um DU-Caminho ($A_1A_2A_3A_4A_5A_2A_7$); essa seqüência deve ser substituída por $A_1A_2A_7$ para que o conjunto de caminhos satisfaça esse critério.

Para satisfazer o Critério INT-Todos-DU-Caminhos, deve-se exercitar todos os DU-Caminhos do programa que tenham influência interprocedimental. Desta forma, o conjunto de caminhos anterior não é suficiente para satisfazer o Critério INT-Todos-DU-Caminhos, os elementos requeridos número 4, 6, 7, 10 e 11 não estão sendo satisfeitos. Para que esses elementos requeridos sejam satisfeitos deve-se incluir os seguintes caminhos:

```
1-  A1 A2 A7 B1 B3 B4 B6 B7 A8 A9 A11 A12 A14
2-  A1 A2 A7 B1 B3 B4 B6 B7 A8 A9 A11 A13 A14
3-  A1 A2 A7 B1 B3 B4 B5 B7 A8 A10 A11 A12 A14
4-  A1 A2 A7 B1 B3 B4 B5 B7 A8 A10 A11 A13 A14
```

O exemplo a seguir mostra as associações requeridas pelo Critérios INT-Todos-Potenciais-Usos para o programa exemplo apresentado por Harrold e Soffa [HS91], reproduzido no Capítulo 2. A abordagem proposta por Harrold e Soffa estabelece as associações de fluxo de dados interprocedimentais para o programa como um todo, propagando a influência de uma variável definida em um módulo, para os outros módulos alcançáveis a partir dele. Os Critérios Potenciais Usos de Integração estão apoiados numa estratégia de teste dois-a-dois; o que implica que não é possível garantir-se que as influências de fluxo de dados indiretas serão exercitadas. No entanto, mostra-se no exemplo a seguir que cria-se a oportunidade para que essas influências indiretas sejam exercitadas, sem promover uma sobrecarga de processamento e incorrer numa possível explosão combinatória no número de elementos requeridos.

Considerando-se a reprodução dos grafos de programa dos módulos *GetMax* e *PairMax* da Figura 4.2, segue a lista de associações exigidas pelo Critério INT-Todos-Potenciais-Usos, para cada uma das conexões entre *GetMax* e *PairMax*.

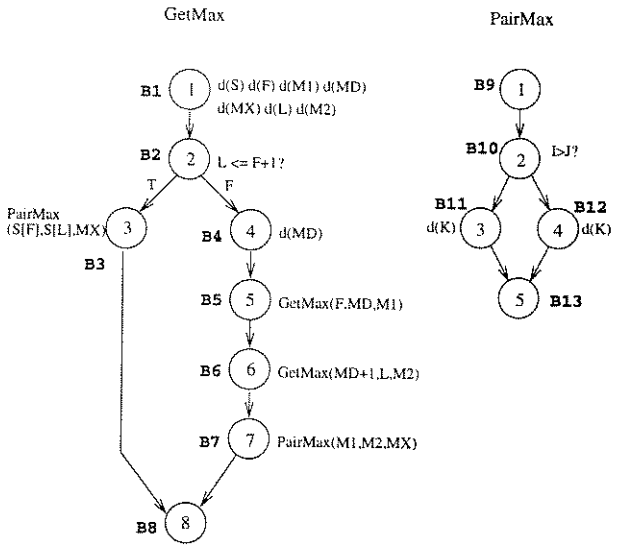

Figura 4.2: Exemplo para Comparar a Abordagem de Harrold e Soffa com os Critérios Potenciais Usos de Integração.

Para a conexão entre *GetMax* e *PairMax* com a chamada no nó 3:

```
De GetMax para PairMax:
1- <1,(2,3),{S,MX}>  <1,(2,3),{I,J,K}>
2- <1,(2,3),{S,MX}>  <1,(2,4),{I,J,K}>

De PairMax para GetMax:
3- <3,(3,5),{K}>  <3,(3,8),{S,F,L,MX}>
4- <4,(4,5),{K}>  <3,(3,8),{S,F,L,MX}>
```

Para a conexão entre *GetMax* e *PairMax* com a chamada no nó 7:

```
De GetMax para PairMax:
5-  <1,(6,7),{M1,M2,MX}>  <1,(2,3),{I,J,K}>
6-  <4,(6,7),{MD}>        <1,(2,3),{I,J,K}>
7-  <5,(6,7),{M1}>        <1,(2,3),{I,J,K}>
8-  <6,(6,7),{M2}>        <1,(2,3),{I,J,K}>
9-  <1,(6,7),{M1,M2,MX}>  <1,(2,4),{I,J,K}>
10- <4,(6,7),{MD}>        <1,(2,4),{I,J,K}>
11- <5,(6,7),{M1}>        <1,(2,4),{I,J,K}>
12- <6,(6,7),{M2}>        <1,(2,4),{I,J,K}>

De PairMax para GetMax:
13-  <3,(3,5),{K}>  <7,(7,8),{MX}>
14-  <4,(4,5),{K}>  <7,(7,8),{MX}>
```

Para a conexão entre *GetMax* e *GetMax* com a chamada no nó 5:

```
De GetMax para GetMax (chamador para chamado):
15- <1,(4,5),{F,M1}>  <1,(2,3),{S,MX,F,L}>
```

```
16- <1,(4,5),{F,M1}>  <1,(4,5),{F,M1}>
17- <1,(4,5),{F,M1}>  <1,(5,6),{L,M2}>
18- <1,(4,5),{F,M1}>  <1,(6,7),{M1,M2,MX}>
19- <1,(4,5),{F,M1}>  <1,(7,8),{F,L,MX}>
20- <4,(4,5),{F,M1}>  <1,(2,3),{S,MX,F,L}>
21- <4,(4,5),{F,M1}>  <1,(4,5),{F,M1}>
22- <4,(4,5),{F,M1}>  <1,(5,6),{L,M2}>
23- <4,(4,5),{F,M1}>  <1,(6,7),{M1,M2,MX}>
24- <4,(4,5),{F,M1}>  <1,(7,8),{F,L,MX}>

De GetMax para GetMax (chamado de volta para chamador):
25- <1,(7,8),{F,L,MX}>    <5,(5,6),{MD}>
26- <3,(3,8),{S,F,L,MX}>  <5,(5,6),{MD}>
27- <5,(7,8),{F}>         <5,(5,6),{MD}>
28- <6,(7,8),{L}>         <5,(5,6),{MD}>
29- <7,(7,8),{MX}>        <5,(5,6),{MD}>
30- <1,(7,8),{F,L,MX}>    <5,(6,7),{M1}>
31- <3,(3,8),{S,F,L,MX}>  <5,(6,7),{M1}>
32- <5,(7,8),{F}>         <5,(6,7),{M1}>
33- <6,(7,8),{L}>         <5,(6,7),{M1}>
34- <7,(7,8),{MX}>        <5,(6,7),{M1}>
35- <1,(7,8),{F,L,MX}>    <5,(7,8),{F}>
36- <3,(3,8),{S,F,L,MX}>  <5,(7,8),{F}>
37- <5,(7,8),{F}>         <5,(7,8),{F}>
38- <6,(7,8),{L}>         <5,(7,8),{F}>
39- <7,(7,8),{MX}>        <5,(7,8),{F}>
```

Para a conexão entre *GetMax* e *GetMax* com a chamada no nó 6:

```
De GetMax para GetMax (chamador para chamado):
40- <1,(5,6),{L,M2}>  <1,(2,3),{S,MX,F,L}>
41- <1,(5,6),{L,M2}>  <1,(4,5),{F,M1}>
42- <1,(5,6),{L,M2}>  <1,(5,6),{L,M2}>
43- <1,(5,6),{L,M2}>  <1,(6,7),{M1,M2,MX}>
44- <1,(5,6),{L,M2}>  <1,(7,8),{F,L,MX}>
45- <4,(5,6),{MD}>  <1,(2,3),{S,MX,F,L}>
46- <4,(5,6),{MD}>  <1,(4,5),{F,M1}>
47- <4,(5,6),{MD}>  <1,(5,6),{L,M2}>
48- <4,(5,6),{MD}>  <1,(6,7),{M1,M2,MX}>
49- <4,(5,6),{MD}>  <1,(7,8),{F,L,MX}>
50- <5,(5,6),{MD}>  <1,(2,3),{S,MX,F,L}>
51- <5,(5,6),{MD}>  <1,(4,5),{F,M1}>
52- <5,(5,6),{MD}>  <1,(5,6),{L,M2}>
53- <5,(5,6),{MD}>  <1,(6,7),{M1,M2,MX}>
54- <5,(5,6),{MD}>  <1,(7,8),{F,L,MX}>

De GetMax para GetMax (chamado de volta para chamador):
55- <1,(7,8),{F,L,MX}>    <6,(6,7),{M2}>
56- <3,(3,8),{S,F,L,MX}>  <6,(6,7),{M2}>
57- <5,(7,8),{F}>         <6,(6,7),{M2}>
```

```
58- <6,(7,8),{L}>          <6,(6,7),{M2}>
59- <7,(7,8),{MX}>         <6,(6,7),{M2}>
60- <1,(7,8),{F,L,MX}>     <6,(7,8),{L}>
61- <3,(3,8),{S,F,L,MX}>   <6,(7,8),{L}>
62- <5,(7,8),{F}>          <6,(7,8),{L}>
63- <6,(7,8),{L}>          <6,(7,8),{L}>
64- <7,(7,8),{MX}>         <6,(7,8),{L}>
```

O conjunto de caminhos necessários para executar essas associações é mostrado a seguir. Os nós com a numeração marcada com "[ ]" representam uma chamada à *GetMax* que não foi expandida na representação do caminho; de forma análoga a marca "{ }" representa uma chamada à *PairMax*:

```
GetMax --> PairMax (3):
1- B1 B2 B3 B9 B10 B11 B13 B8
2- B1 B2 B3 B9 B10 B12 B13 B8

GetMax --> PairMax (7):
3- B1 B2 B4 [B5] [B6] B7 B9 B10 B11 B13 B8
4- B1 B2 B4 [B5] [B6] B7 B9 B10 B12 B13 B8

GetMax --> GetMax (5):
5- B1 B2 B4 B5 B1 B2 B3 B8 [B6] {B7} B8
6- B1 B2 B4 B5 B1 B2 B4 B5 [B6] {B7} B8 [B6] {B7} B8

GetMax --> GetMax (6):
7- B1 B2 B4 [B5] B6 B1 B2 B3 B8 {B7} B8
8- B1 B2 B4 [B5] B6 B1 B2 B4 [B5] [B6] {B7} B8 {B7} B8
```

Harrold e Soffa exigem as associações envolvendo as definições de K nos nós B_{11} e B_{12} e os usos de K em (B_{10}, B_{11}). Só é possível cobrir essas associações através da execução de *PairMax* pela chamada no nó B_7 dentro de uma chamada recursiva de *GetMax*; desta forma, ao retornar da execução e executar novamente *PairMax* pela chamada do nó B_7 cobrir-se-ia as associações exigidas. Os caminhos {3, 4, 5, 6, 7, 8}, definidos para cobrir as associações exigidas pelo Critério INT-Todos-Potenciais-Usos, podem levar à execução das associações definidas por Harrold e Soffa, apesar de isso não ser garantido pelo critério. A principal diferença entre os Critérios Potenciais Usos e o de Harrold e Soffa é que o primeiro é aplicado a um par de módulos de cada vez, enquanto o segundo é aplicado a todo o programa de uma única vez. Dessa difereça surgem situações como a apontada no exemplo anterior, onde uma abordagem exige associações envolvendo vários módulos (incluindo chamadas recursivas) que não são exigidas pela outra abordagem.

4.3 Análise de Propriedades

Nesta seção são investigadas algumas das propriedades dos Critérios Potenciais Usos de Integração, apresentados na seção anterior. Inicialmente é considerada a relação de inclusão entre os critérios, em seguida é estabelecida uma hierarquia de critérios de integração baseada na sua relativa habilidade de detectar defeitos em um programa.

4.3.1 Relação de Inclusão

A hierarquia de inclusão do Capítulo 2 mostra a relação entre critérios de teste para o teste de unidade; nesta seção a mesma relação é analisada para critérios de teste de integração. Inicialmente consideram-se alguns Critérios de Teste baseados em Análise de Fluxo de Controle e em seguida são analisados os Critérios Baseados em Análise de Fluxo de Dados.

Teorema 4.1 *O critério* Todas-Relações *inclui estritamente o critério* Todos-Módulos.

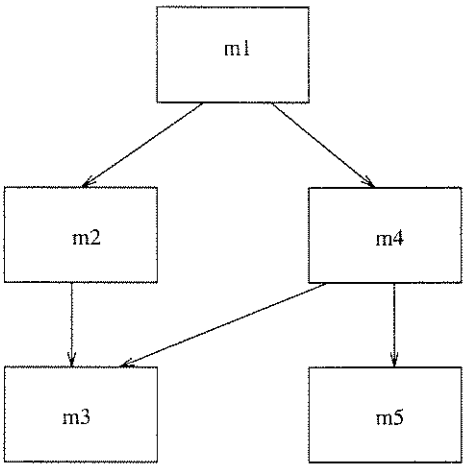

Figura 4.3: Grafo de Chamada CG

Prova: Inicialmente prova-se que *Todas-Relações* $\stackrel{\text{inclui}}{\Longrightarrow}$ *Todos-Módulos.* Supondo que o grafo de chamada é "bem-formado", todo módulo é alcançável do módulo de entrada e, conseqüentemente, todo módulo faz parte de uma conexão (M_1, M_2). Suponha que o par (CG, Ψ) satisfaz *Todas-Relações*; para toda conexão (M_1, M_2) em CG existe um caminho em Ψ que inclui um arco de M_1 para M_2, desta forma todo par de módulos do programa será executado; como todo módulo é parte de um par, então todo módulo será executado.

Agora, prova-se que *Todos-Módulos* $\stackrel{\text{inclui}}{\not\Longrightarrow}$ *Todas-Relações.* Considere o grafo de chamada CG da Figura 4.3. O par (CG, Ψ) satisfaz *Todos-Módulos*, onde Ψ=$\{(m_1, m_2, m_3), (m_1, m_4, m_5)\}$. Esse par não satisfaz *Todas-Relações* pois Ψ não contém a relação (m_4, m_3). Portanto, *Todos-Módulos* não inclui *Todas-Relações.*

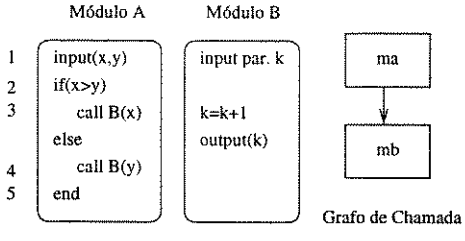

Figura 4.4: Programa Exemplo # 1.

Teorema 4.2 *O critério* Todas-Relações-Múltiplas *inclui estritamente o critério* Todas-Relações.

Prova: Inicialmente prova-se que *Todas-Relações-Múltiplas* inclui *Todas-Relações.* Suponha que o par (CG, Ψ) satisfaz *Todas-Relações-Múltiplas* Ψ inclui um subcaminho que executa todo arco e entre M_1 e M_2 para toda conexão (M_1, M_2) em CG, como toda conexão é coberta (CG, Ψ) satisfaz *Todas-Relações.* Agora, prova-se que *Todas-Relações* não inclui *Todas-Relações-Múltiplas.* Considere o programa da Figura 4.4, suponha que o par (CG, Ψ) satisfaz *Todas-Relações* e Ψ=$\{(m_a, m_b)\}$ inclui o comando "*call B*" no nó 3 do módulo A. Como "*call B*" no nó 4 não é executado (CG, Ψ) não satisfaz *Todas-Relações-Múltiplas.*

O próximo passo é analisar os Critérios de Teste de Integração baseados em Análise de Fluxo de Dados. Esses critérios são incomparáveis com os critérios baseados em análise de fluxo de controle, anteriormente apresentados. O critério *INT-Todas-Definições* é incomparável ao critério *Todas-Relações-Múltiplas* já que *INT-Todas-Definições* deve ser aplicado a um par de módulos de cada vez enquanto *Todas-Relações-Múltiplas* considera o programa como um todo. Entretanto, pode-se provar, como se segue, algumas relações entre os critérios baseados em análise de fluxo de dados e os baseados em análise de fluxo de controle considerando-se que os critérios baseados em análise de fluxo de dados são aplicados a todo par de módulos no programa.

Teorema 4.3 *O critério* INT-Todas-Definições *e o critério* Todas-Relações-Múltiplas *são incomparáveis.*

Prova: Inicialmente prova-se que *INT-Todas-Definições* $\overset{\text{inclui}}{\not\Longrightarrow}$ *Todas-Relações-Múltiplas.*

Assume-se que o conjunto in_c para todo par de módulos (M_1, M_2) e toda chamada c entre esses módulos tem pelo menos um elemento; isto é, cada comando de chamada deve envolver pelo menos uma variável de comunicação. Seja $in_c = X$, analisa-se o caso onde não existe uso de $formal_c(x)$ em M_2 e não existe uso de x alcançável a partir do nó da chamada c em M_1. Desta forma o Critério *INT-Todas-Definições* não vai exigir associações entre M_1 e M_2; assim o arco (M_1, M_2) não será executado. Conclui-se que *INT-Todas-Definições* não inclui *Todas-Relações-Múltiplas.* O contrário também não é válido; pode-se exercitar todas as chamadas e não satisfazer *INT-Todas-Definições*, veja o exemplo da Figura 4.5, o caminho $(n_s, 1_a, 2_a, 4_a, 6_a, n_{in}, 1_b, 3_b, 4_b, n_{out}, 7_a, n_f)$ satisfaz o critério *Todas-Relações-Múltiplas*, mas não satisfaz *INT-Todas-Definições*; já que a associação entre a definição de x no nó 3_a e seu uso em 3_b não é coberta. Portanto *INT-Todas-Definições* e *Todas-Relações-Múltiplas* são incomparáveis.

Teorema 4.4 *O Critério* INT-Todos-Usos *é incomparável ao Critério* Todos-Relações-Múltiplas.

Prova: A prova é análoga à do Teorema 4.3.

Teorema 4.5 *Se existir pelo menos uma variável de comunicação em uma chamada c de M_1 para M_2 sempre vai existir pelo menos uma potencial associação interprocedimental entre M_1 e M_2.*

Prova: Assume-se que não existem anomalias de fluxo de dados, ou seja, toda variável usada foi definida e toda definição de variável alcança pelo menos um uso. Supondo que x é a variável de comunicação presente na chamada c de M_1 para M_2, então $x \in in_c$ e/ou $x \in out_c$. Se $x \in in_c$ existe uma definição $d_{n_1}(M_1, x)$ que alcança n_c por um sub-caminho p livre de definição c.r.a x, caso contrário x estaria indefinida na chamada c configurando uma anomalia de fluxo de dados, $formal_c(x)$ definida em $n_{in}(M_2)$ alcança pelo menos um potencial uso de $formal_c(x)$ em n_2 por um sub-caminho q livre de definição c.r.a $formal_c(x)$; desta forma, uma potencial associação interprocedimental de chamada é estabelecida, exigindo um sub-caminho de tipo $(n_1) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n_2)$. No caso de $x \in out_c$ existe uma definição $d_{n_2}(M_2, formal_c(x))$ que alcança $n_{out}(M_2)$ por um sub-caminho p livre de definição c.r.a $formal_c(x)$ e um potencial uso de x em $n_1 \in M_1$ alcançado a partir de n_c por um sub-caminho p livre de definição c.r.a x; desta forma, seria estabelecida uma potencial associação interprocedimental de retorno exigindo a execução de um sub-caminho do tipo $(n_2) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (n_1)$. Portanto, se existir pelo menos uma variável de comunicação em uma chamada c de M_1 para M_2, sempre vai existir pelo menos uma potencial associação interprocedimental entre M_1 e M_2.

Teorema 4.6 *O critério* INT-Todos-Potenciais-Usos *inclui estritamente o critério* Todas-Relações-Múltiplas.

Prova: Inicialmente prova-se que *INT-Todos-Potenciais-Usos* $\overset{\text{inclui}}{\Longrightarrow}$ *Todas-Relações-Múltiplas.*

Segue da prova do Teorema 4.3 que os critérios que exigem a ocorrência explícita do uso da variável para estabelecer requisitos de teste, como é o caso de *INT-Todas-Definições* e *INT-Todos-Usos* não incluem o Critério *Todas-Relações-Múltiplas.* O Critério *INT-Todos-Potenciais-Usos* por sua vez exige associações de fluxo de dados mesmo não existindo o uso da variável; desta forma, considerando-se a existência de pelo menos uma variável de comunicação para cada chamada c entre cada par de módulos (M_1, M_2) do programa, sempre vai existir pelo menos uma associação entre M_1 e M_2 pela chamada c (Teorema 4.5), portanto *INT-Todos-Potenciais-Usos* inclui *Todas-Relações-Múltiplas.*

Teorema 4.7 *O critério* INT-Todos-C-Usos/Alguns-P-Usos *inclui estritamente o critério* INT-Todas-Definições *e o critério* Todos-P-Usos/Alguns-C-Usos *inclui estritamente o critério* INT-Todas-Definições.

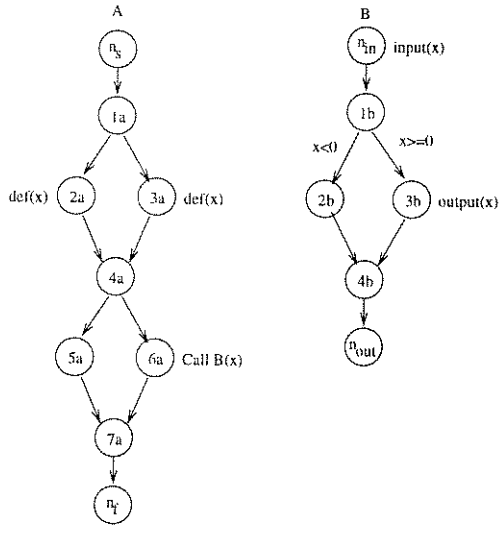

Figura 4.5: Programa Exemplo # 2.

Prova: Inicialmente prova-se que *INT-Todos-C-Usos/Alguns-P-Usos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todas-Definições*. Suponha que a tripla (M_1, M_2, Φ) satisfaz o critério *INT-Todos-C-Usos/Alguns-P-Usos*. Seja $d_m(M_1, x)$ uma definição, e c-uso$_n(M_2, formal_c(x))$ (ou p-uso$_{(n,n')}(M_2, formal_c(x))$) o c-uso (p-uso) alcançável por $d_m(M_1, x)$ para todo $x \in in_c$ e seja $d_n(M_2, formal_c(x))$ uma definição, e c-uso$_n(M_1, x)$ (p-uso$_{(m,m')}(M_1, x)$) o c-uso (p-uso) alcançável por $d_n(M_2, formal_c(x))$ para todo $x \in out_c$. Todas essas definições alcançam um c-uso ou se não existe o *c-uso* alcançam um p-uso, conseqüentemente, toda definição é exercitada em conjunto com um de seus usos, portanto *INT-Todos-C-Usos/Alguns-P-Usos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todas-Definições*. A prova de que *INT-Todos-P-Usos/Alguns-C-Usos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todas-Definições* é similar.

Agora prova-se que *INT-Todas-Definições* $\stackrel{\text{inclui}}{\not\Longrightarrow}$ *INT-Todos-C-Usos/Alguns-P-Usos* e *INT-Todas-Definições* $\stackrel{\text{inclui}}{\not\Longrightarrow}$ *INT-Todos-P-Usos/Alguns-C-Usos*

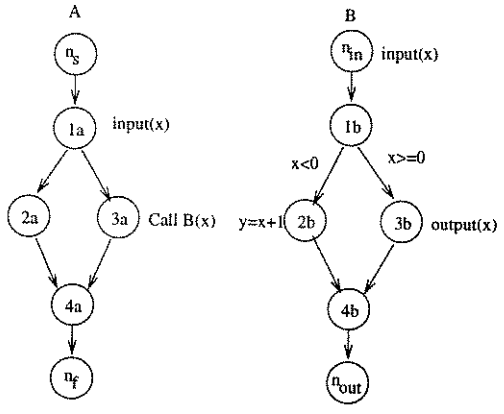

Figura 4.6: Programa Exemplo # 3.

Considere o programa da Figura 4.6, a tripla (A, B, Φ) com $\Phi = \{(n_s, 1_a, 3_a, n_{in}, 1_b, 3_b, 4_b, n_{out}, 4_a, n_f)\}$ satisfaz *INT-Todas-Definições* mas não satisfaz *INT-*

Todos-C-Usos/Alguns-P-Usos ou *INT-Todos-P-Usos/Alguns-C-Usos*

Teorema 4.8 *O critério* INT-Todos-Usos *inclui estritamente o critério* INT-Todos-C-Usos/-Alguns-P-Usos *e o critério* INT-Todos-Usos *inclui estritamente o critério* INT-Todos-P-Usos/Alguns-C-Usos.

Prova: Inicialmente prova-se que *INT-Todos-Usos* inclui *INT-Todos-C-Usos/-Alguns-P-Usos*. Suponha que a tripla (M_1, M_2, Φ) satisfaz *INT-Todos-Usos*. Para toda definição $d_m(M_1, x)$ com $x \in in_c$, Φ inclui um sub-caminho livre de definição de $d_m(M_1, x)$ para todo c-uso$_n(M_2, formal_c(x))$ e todo p-uso$_{(n,n')}$ $(M_2, formal_c(x))$. Para toda definição $d_n(M_2, formal_c(x))$ com $x \in out_c$, Φ inclui um sub-caminho livre de definição de $d_n(M_2, formal_c(x))$ para todo c-uso$_m(M_1, x)$, e todo p-uso$_{(m,m')}(M_1, x)$. Como os usos considerados incluem todo c-uso e todo p-uso, *INT-Todos-Usos* $\overset{\text{inclui}}{\Longrightarrow}$ *INT-Todos-C-Usos/Alguns-P-Usos* e *INT-Todos-Usos* $\overset{\text{inclui}}{\Longrightarrow}$ *INT-Todos-P-Usos/Alguns-C-Usos*. Agora, prova-se que *INT-Todos-C-Usos/Alguns-P-Usos* $\overset{\text{inclui}}{\not\Longrightarrow}$ *INT-Todos-Usos* e *INT-Todos-P-Usos/Alguns-C-Usos* $\overset{\text{inclui}}{\not\Longrightarrow}$ *INT-Todos-Usos*.

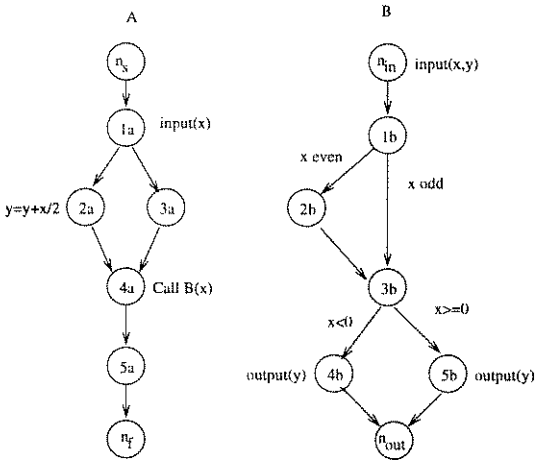

Figura 4.7: Programa Exemplo # 4.

Considere o programa da Figura 4.7. A tripla (A, B, Φ) satisfaz *INT-Todos-P-Usos/Alguns-C-Usos*, para $\Phi = \{(n_s, 1_a, 3_a, 4_a, n_{in}, 1_b, 3_b, 5_b, n_{out}, 5_a, n_f), (n_s, 1_a, 2_a, 4_a, n_{in}, 1_b, 2_b, 3_b, 4_b, n_{out}, 5_a, n_f)\}$, mas não *INT-Todos-Usos*, já que não é incluído um sub-caminho livre de definição da definição de y no nó 2_a ao c-uso de y no nó 5_b. A tripla (A, B, Φ) satisfaz *INT-Todos-C-Usos/Alguns-P-Usos* para $\Phi = \{(n_s, 1_a, 2_a, 4_a, n_{in}, 1_b, 3_b, 5_b, n_{out}, 5_a, n_f), (n_s, 1_a, 2_a, 4_a, n_{in}, 1_b, 3_b, 4_b, n_{out}, 5_a, n_f)\}$ mas não *INT-Todos-Usos*, já que não é incluído um sub-caminho livre de definição da definição de x no nó 1_a ao p-uso de x no arco $(1_b, 2_b)$.

Teorema 4.9 *O critério* INT-Todos-DU-Caminhos *inclui estritamente o critério* INT-Todos-Usos.

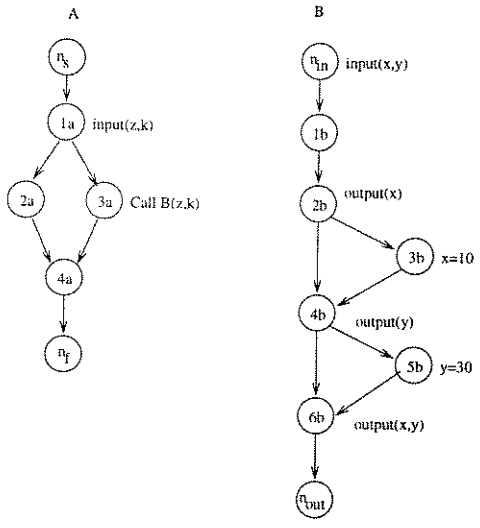

Figura 4.8: Programa Exemplo # 5.

Prova: Inicialmente prova-se que *INT-Todos-DU-Caminhos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todos-Usos*. Suponha que a tripla (M_1, M_2, Φ) satisfaz *INT-Todos-DU-Caminhos*; para que Φ satisfaça *INT-Todos-Usos* Φ deve incluir para cada chamada c de M_1 para M_2, pelo menos um sub-caminho de cada uma das seguintes formas: Para toda variável $x \in in_c$.

- $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n)$, para cada $d_m(M_1, x)$, c-uso$_n(M_2, formal_c(x))$, com p livre de definição c.r.a x e q livre de definição c.r.a $formal_c(x)$.
- $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n, n')$, para cada $d_m(M_1, x)$, p-uso$_{(n,n')}(M_2, formal_c(x))$, com p livre de definição c.r.a x e $q\cdot(n)$ livre de definição c.r.a $formal_c(x)$.

Para toda variável $x \in out_c$.

- $(n)\cdot p\cdot(n_{out}(M_2))\cdot q\cdot(m)$, para cada $d_n(M_2, formal_c(X))$, c-uso$_m(M_1, x)$, com p livre de definição c.r.a $formal_c(x)$ e q livre de definição c.r.a x.
- $(n)\cdot p\cdot(n_{out}(M_2))\cdot q\cdot(m, m')$, para cada $d_n(M_2, formal_c(X))$, p-uso$_{(m,m')}$ (M_1, x), com p livre de definição c.r.a $formal_c(x)$ e $q\cdot(m)$ livre de definição c.r.a x.

Como Φ satisfaz *INT-Todos-DU-Caminhos*, Φ contém caminhos que percorrem DU-Caminhos para cada um dos tipos de sub-caminhos acima, desta forma *INT-Todos-DU-Caminhos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todos-Usos*.

Agora, prova-se que *INT-Todos-Usos* $\stackrel{\text{inclui}}{\not\Longrightarrow}$ *INT-Todos-DU-Caminho*.

Considere o programa da Figura 4.8. A tripla (A, B, Φ) com $\Phi = \{(n_s, 1_a, 3_a, n_{in}, 1_b, 2_b, 3_b, 4_b, 5_b, 6_b, n_{out}, 4_a, n_f), (n_s, 1_a, 3_a, n_{in}, 1_b, 2_b, 4_b, 6_b, n_{out}, 4_a, n_f)\}$ satisfaz *INT-Todos-Usos* mas não satisfaz *INT-Todos-DU-Caminho*, pois Φ não contém todos os sub-caminhos da forma $(1_a)\cdot p\cdot(3_a)\cdot(n_{in}(B))\cdot q\cdot(n_6)$, onde p é livre de definição c.r.a k e q é livre de definição c.r.a y. Em particular Φ não inclui o sub-caminho $(1_a, 3_a, n_{in}, 1_b, 2_b, 3_b, 4_b, 6_b)$.

Teorema 4.10 *O critério* INT-Todos-DU-Caminhos *não inclui o critério* INT-Todos-Potenciais-Usos e INT-Todos-DU-Caminhos *não inclui o critério* INT-Todos-Potenciais-Usos/DU.

Prova: Considere o programa da Figura 4.9. A tripla (A, B, Φ) com $\Phi = \{(n_s, 1_a, 3_a, n_{in}, 1_b, 2_b, 4_b, 5_b, 6_b, n_{out}, 4_a, n_f), (n_s, 1_a, 3_a, n_{in}, 1_b, 3_b, 4_b, 5_b, 6_b, n_{out}, 4_a, n_f)\}$ satisfaz *INT-Todos-DU-Caminhos* mas não satisfaz *INT-Todos-Potenciais-Usos* ou *INT-Todos-Potenciais-Usos/DU* já que o Φ não inclui um sub-caminho livre de definição da definição de x no nó 1_a para o potencial uso no nó 6_b (ou um DU-Caminho para satisfazer o critério *INT-Todos-Potenciais-Usos/DU*).

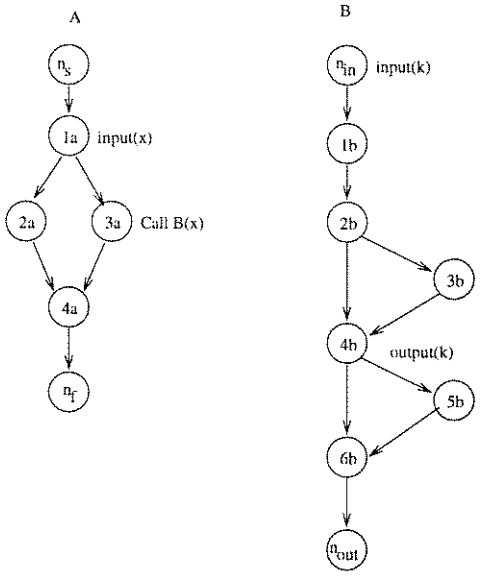

Figura 4.9: Programa Exemplo # 6.

Teorema 4.11 *O critério* INT-Todos-Potenciais-Usos *inclui estritamente o critério* INT-Todos-Usos.

Prova: Inicialmente prova-se que *INT-Todos-Potenciais-Usos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todos-Usos*. Suponha que a tripla (M_1, M_2, Φ) satisfaz *INT-Todos-Potenciais-Usos*, deve-se mostrar que Φ inclui para cada chamada c de M_1 para M_2 pelo menos um sub-caminho de cada uma das seguintes formas:

Para toda variável $x \in in_c$.

- $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n)$, para cada $d_m(M_1, x)$, c-uso$_n(M_2, formal_c(x))$, com p livre de definição c.r.a x e q livre de definição c.r.a $formal_c(x)$.
- $(m)\cdot p\cdot(n_c)\cdot(n_{in}(M_2))\cdot q\cdot(n, n')$, para cada $d_m(M_1, x)$, p-uso$_{(n,n')}(M_2, formal_c(x))$, com p livre de definição c.r.a x e $q\cdot(n)$ livre de definição c.r.a $formal_c(x)$.

Para toda variável $x \in out_c$.

- $(n)\cdot p\cdot(n_{out}(M_2))\cdot q\cdot(m)$, para cada $d_n(M_2, formal_c(x))$, c-uso$_m(M_1, x)$, com p livre de definição c.r.a $formal_c(x)$ e q livre de definição c.r.a x.

- $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m, m')$, para cada $d_n(M_2, formal_c(x))$, p-uso$_{(m,m')}$ (M_1, x), com p livre de definição c.r.a $formal_c(x)$ e $q \cdot (m)$ livre de definição c.r.a x.

Como Φ satisfaz *INT-Todos-Potenciais-Usos*, e como todo uso de variável é também um potencial uso, Φ inclui sub-caminhos de cada um dos tipos de sub-caminhos acima; desta forma, *INT-Todos-Potenciais-Usos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todos-Usos.*

Para provar que *INT-Todos-Potenciais-Usos* inclui estritamente *INT-Todos-Usos* basta mostrar que uma tripla (M, N, Φ) satisfazendo *INT-Todos-Usos* não vai necessariamente incluir sub-caminhos da forma $(m_i) \cdot p \cdot (n_c) \cdot (n_{in}(N)) \cdot q \cdot (n, n')$ ou $(n_i) \cdot p \cdot (n_{out}(N)) \cdot q \cdot (m, m')$, isso ocorre se não houver uso explícito da variável nos nós n ou m, no entanto esses sub-caminhos são requeridos pelo critério *INT-Todos-Potenciais-Usos* se a variável estiver viva nos nós n ou m.

Teorema 4.12 *O critério* INT-Todos-Potenciais-Usos/DU *inclui estritamente* INT-Todos-Potenciais-Usos.

Prova: A prova é similar à prova de que o critério *INT-Todos-DU-Caminhos* $\stackrel{\text{inclui}}{\Longrightarrow}$ *INT-Todos-Usos.* Informalmente, como *INT-Todos-Potenciais-Usos/DU* requer somente que o sub-caminho escolhido para cobrir a potencial associação seja um Potencial DU-Caminho, esse sub-caminho também faz parte do conjunto de sub-caminhos requeridos para satisfazer *INT-Todos-Potenciais-Usos.*

Teorema 4.13 *O critério* INT-Todos-Potenciais-DU-Caminhos *inclui estritamente o critério* INT-Todos-Potenciais-Usos/DU.

Prova: Inicialmente prova-se que o critério *INT-Todos-Potenciais-DU-Caminhos* inclui o critério *INT-Todos-Potenciais-Usos/DU.*

A prova decorre da definição dos critérios, *INT-Todos-Potenciais-DU-Caminhos* exige todos os Potenciais DU-Caminhos entre dois módulos enquanto *INT-Todos-Potenciais-Usos/DU* só exige um Potencial DU-Caminho para cobrir uma associação entre dois módulos.

Teorema 4.14 *O critério* INT-Todos-Potenciais-DU-Caminhos *inclui estritamente o critério* INT-Todos-DU-Caminhos.

Prova: A prova também decorre diretamente da definição dos critérios. Informalmente, como todo *DU-Caminho* requerido por *INT-Todos-DU-Caminhos* é também um Potencial DU-Caminho requerido por *INT-Todos-Potenciais-DU-Caminhos* a relação de inclusão é válida. No entanto nem todo DU-Caminho é um Potencial-DU-Caminho; portanto, a inclusão é estrita.

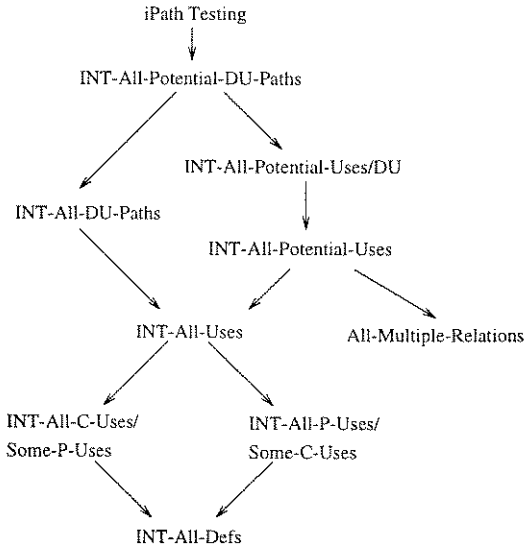

Figura 4.10: Relação de Inclusão para Integração.

Teorema 4.15 *O critério* Todos-iCaminhos *inclui estritamente o critério* INT-Todos-Potenciais-DU-Caminhos.

> *Prova*: Essa relação é verdadeira já que o critério *Todos-iCaminhos* pede que todos os caminhos entre um módulo M_1 e M_2 sejam exercitados, incluindo aqueles DU-Caminhos que envolvem associações interprocedimentais. Os *iCaminhos* que envolvem laços não são requeridos pelo Critério *INT-Todos-Potenciais-DU-Caminhos*; portanto a inclusão é estrita.

O conjunto de relações de inclusão apresentados nesta seção é graficamente resumido pela hierarquia mostrada na Figura 4.10. A relação de inclusão obtida para os critérios de integração mantém a mesma hierarquia que os correspondentes critérios de unidade. Essa propriedade era esperada de acordo com a definição dos critérios de integração e pôde ser comprovada pelos teoremas apresentados nesta seção.

A análise apresentada anteriormente considera os Critérios Potenciais Usos de Integração Básicos; segue a análise para os Critérios Potenciais Usos de Integração Executáveis.

A análise de inclusão, conduzida por Frankl e Weyuker [FW88] para os critérios da Família de Critérios de Fluxo de Dados (DFCF), aplica-se à classe de programas P que satisfazem as propriedades NSUP - "No Syntatic Undefined P-Use Property" e NSL - "No Straight Line Property". Frankl e Weyuker observam que, na presença de caminhos não executáveis nenhum critério da DFCF preenchia as propriedades de um "bom" critério de teste; por exemplo, nenhum deles incluia o critério todos-arcos. Restringindo a análise à classe de programas que satisfazem NFUP - "No Feasible Undefined P-Uses Property" esse problema é contornado. O problema é que é indecidível se um dado programa satisfaz essa propriedade. Outra alternativa seria restringir a análise aos programas que satisfazem a propriedade NA - "No Anomalies" o que, segundo Frankl e Weyuker, seria muito restritivo já que bons programas não satisfazem necessariamente essa propriedade. Segundo Maldonado [Mal91], para os Critérios Potenciais Usos basta requerer a propriedade LDEN - "At Least one Definition in the Entry Node" – a definição de pelo menos uma variável de P no nó de entrada. Com a ressalva que deve haver pelo menos uma definição de uma variável de comunicação para cada chamada c em cada módulo do programa, a mesma propriedade

é exigida para os Critérios Potenciais Usos de Integração. Observa-se que esta propriedade é facilmente preenchida pela maioria dos programas.

Teorema 4.16 *O critério* (INT-Todos-Potenciais-Usos/DU)* *inclui estritamente o critério* (Todas-Relações-Múltiplas)*.

Prova: Suponha que T é *(INT-Todos-Potenciais-Usos/DU)**-adequado para P. Seja n_c uma chamada executável de A para B. Uma vez que a chamada é executável existe pelo menos um caminho interprocedimental executável $\varphi = (n_{in}(A)) \cdot p \cdot (n_c) \cdot (n_{in}(B)) \cdot q \cdot (n_{out}(B)) \cdot r \cdot (n_{out}(A))$ tal que a chamada n_c é incluída em φ. 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_a para o nó n_c e a partir do nó $n_{in}(B)$ para algum nó n_b, para alguma variável v e $formal_c(v)$, respectivamente; um desses Potenciais DU-Caminhos executáveis será incluído em φ pois T é *(INT-Todos-Potenciais-Usos/DU)**-adequado; desta forma a chamada n_c será incluída em φ. A segunda parte é trivial já que mesmo que $formal_c(v)$ seja redefinida no nó seguinte a $n_{in}(B)$ vai existir um Potencial DU-Caminho entre esses dois nós.

- Considere-se que o caminho $\varphi_i = ((n_{in}(A)) \cdot p \cdot (n_c))$ seja um caminho livre de laço executável. Uma vez que o programa P satisfaz a propriedade LDEN, o nó $n_{in}(A)$ possui pelo menos uma definição de alguma variável v. Se os nós em p não tiverem redefinição da variável v definida no nó $n_{in}(A)$, então o caminho $(n_{in}(A) \cdot p \cdot n_c)$ é um Potencial DU-Caminho executável c.r.a v do nó $n_{in}(A)$ para o nó n_c. Se algum nó n_d em p tiver uma redefinição de v, então o caminho $((n_d) \cdot (n_c))$ sub-caminho de $((n_{in}(A)) \cdot p \cdot (n_c))$ é um Potencial DU-Caminho executável c.r.a v do nó n_d ao nó n_c.
- Considere-se que o caminho $\varphi_i = ((n_{in}(A)) \cdot p \cdot (n_c))$ não seja um caminho livre de laços. Seja $((n_l) \cdot s \cdot (n_l))$ o último laço no caminho φ_i antes da ocorrência do nó n_c; isto é, $\varphi_i = (n_{in}(A) \cdot p' \cdot (n_l) \cdot s \cdot (n_l) \cdot p'' \cdot (n_c))$. O caminho $\varphi_{i_1} = ((n_l) \cdot p'' \cdot (n_c))$ é um caminho livre de laços executável. Se em algum nó n_d do caminho φ_{i_1} ocorrer a definição de uma variável v, é imediata a conclusão de que o caminho φ_{i_1} inclui um Potencial DU-Caminho executável c.r.a v do nó n_d para n_c (parte i). Ainda, é fácil concluir que o caminho $(s \cdot (n_l) \cdot p'' \cdot (n_c))$ é livre de laço executável; adicionalmente um dos nós em s deve possuir uma definição para alterar a condição do laço; desta forma, $(s \cdot (n_l) \cdot p'' \cdot (n_c))$ inclui um Potencial DU-Caminho executável (parte i). Se a definição ocorrer no nó n_l, tem-se simplesmente o caso de um caminho $\varphi_{i_1} = ((n_l) \cdot p'' \cdot (n_c))$ com uma definição em n_l.

Dada a prova acima é trivial mostrar que *(INT-Todos-Potenciais-Usos)** inclui *(Todas-Relações-Múltiplas)**, já que não se exige um caminho livre de laço para cobrir a associação; basta que para cada nó de chamada executável exista pelo um caminho que o inclua; no pior caso, sempre haverá pelo menos uma definição de variável no nó inicial (LDEN), dando

origem a uma associação do nó inicial para o nó n_c; caso haja uma redefinição da variável no nó n_d, teria-se a associação entre n_d e o nó de chamada. Da mesma forma, também é trivial mostrar que *(Todos-iCaminhos)** inclui *(INT-Todos-Potenciais-DU-Caminhos)** e que este, por sua vez, inclui *(INT-Todos-Potenciais-Usos/DU)**. Essas relações são resumidas na Figura 4.11.

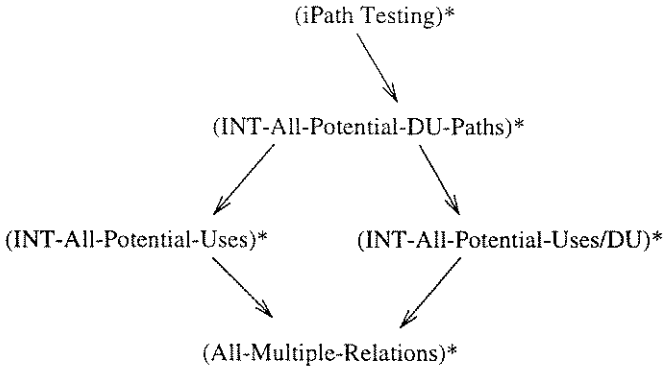

Figura 4.11: Relação de Inclusão dos Critérios de Integração Executáveis.

4.3.2 Habilidade na Detecção de Defeitos

Da mesma forma que no Capítulo 3, a habilidade de detectar defeitos dos Critérios Potenciais Usos para o teste de unidade foi comparada com a de outros critérios de teste baseados em análise de fluxo de dados e com os Critérios Todos-Caminhos e Todas-Decisões, os Critérios Potenciais Usos para o teste de integração também devem ser estudados segundo essa relação.

No Capítulo 3 demonstrou-se que os Critérios Potenciais Usos mantêm uma hierarquia de critérios entre o teste de caminhos e o teste de ramos, mesmo quando sua habilidade de detectar defeitos é considerada. Esta seção mostra que a mesma relação é mantida quando os correspondentes critérios de integração são analisados. Para tanto, considera-se o Critério Todas-Relações ("All-Relations"), proposto por Linnenkugel e Müllerburg [LM90], como representante do teste de ramos para o teste de integração; esse critério pede que pelo menos uma chamada entre cada par de módulos de um grafo de chamada seja exercitada. Para representar o teste de caminhos define-se o *Critério Todos-iCaminhos* ("All-iPaths"); considerando-se a definição de $iCAMINHOS(M_1, M_2)$ dada no Capítulo 2, esse critério pede que todos os caminhos entre dois módulos M_1 e M_2 sejam executados pelos casos de teste.

Teorema 4.17 *O critério* INT-Todos-Potenciais-Usos *não cobre propriamente o critério* Todas-Relações.

Prova:

O subdomínio formado pelo critério *Todas-Relações* é composto pela união dos dados de entrada que executam cada chamada no programa. O exemplo da Figura 4.12 mostra o subdomínio C_1 que exercita a chamada c_1 e o subdomínio C_2 que exercite a chamada c_2; o subdomínio para o critério *Todas-Relações* é dada por $C_1 \cup C_2$. Supondo que cada chamada c envolve pelo

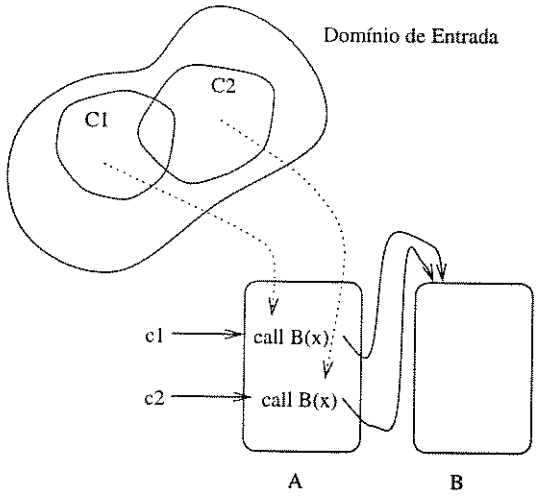

Figura 4.12: Divisão do Domínio de Entrada.

menos uma variável de comunicação, teriamos pelo menos uma associação interprocedimental, envolvendo a chamada c, exigida pelo Critério *INT-Todos-Potenciais-Usos*. Cada subdomínio C correspondente a cada chamada c pode ser dividido em duas partes, os elementos que exercitam alguma associação exigida por *INT-todos-Potenciais-Usos* – C' – e os elementos que não exercitam nenhuma associação – C'' – o subconjunto C'' não pode ser composto pela união de subdomínios de *INT-Todos-Potenciais-Usos*; desta forma, o Critério *INT-Todos-Potenciais-Usos* não cobre o Critério *Todas-Relações*.

As provas de que *INT-Todos-Usos* cobre propriamente *INT-Todas-Definições*, *INT-Todos-Potenciais-Usos* cobre propriamente *INT-Todos-Usos*, e *INT-Todos-Potenciais-DU-Caminhos* cobre propriamente *INT-Todos-DU-Caminhos* são análogas às provas apresentadas no Capítulo 3 para os critérios de unidade. A prova para a relação *INT-Todos-Potenciais-DU-Caminhos Estendido a Ciclos* cobre propriamente *INT-Todos-Potenciais-Usos* também é análoga à prova da relação equivalente para os critérios de unidade.

Teorema 4.18 *O critério* Todos-iCAMINHOS *cobre propriamente o critério* INT-Todos-Potenciais-DU-Caminhos Estendido a Ciclos.

Prova: Todo e qualquer Potencial DU-Caminho Estendido a Ciclo de integração entre os módulos M_1 e M_2 da forma $(m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n)$ ou $(n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m)$ é sub-caminho de um $iCAMINHO(M_1, M_2)$ (caminho completo entre M_1 e M_2) da forma $(n_{in}(M_1)) \cdot k \cdot (m) \cdot p \cdot (n_c) \cdot (n_{in}(M_2)) \cdot q \cdot (n) \cdot j \cdot (n_{out}(M_2)) \cdot l \cdot (n_{out}(M_1))$ ou $(n_{in}(M_1)) \cdot k \cdot (n_c) \cdot (n_{in}(M_2)) \cdot j \cdot (n) \cdot p \cdot (n_{out}(M_2)) \cdot q \cdot (m) \cdot l \cdot (n_{out}(M_1))$. Como representado na Figura 4.13 o subdomínio definido pelo sub-caminho é a união dos subdomínios definidos pelos $iCAMINHOS(M_1, M_2)$; portanto, $SD_{INT-Todos-PDU-Est-Ciclo}(P, S) \subseteq SD_{Todos-iCAMINHOS}(P, S)$; conseqüentemente, Todos-iCAMINHOS cobre propriamente INT-Todos-Potenciais-DU-Caminhos Estendido a Ciclo.

As provas apresentadas nesta seção refletem-se na hierarquia mostrada na Figura 4.14; desta forma, mostra-se que os Critérios Potenciais Usos de integração definem uma ordem

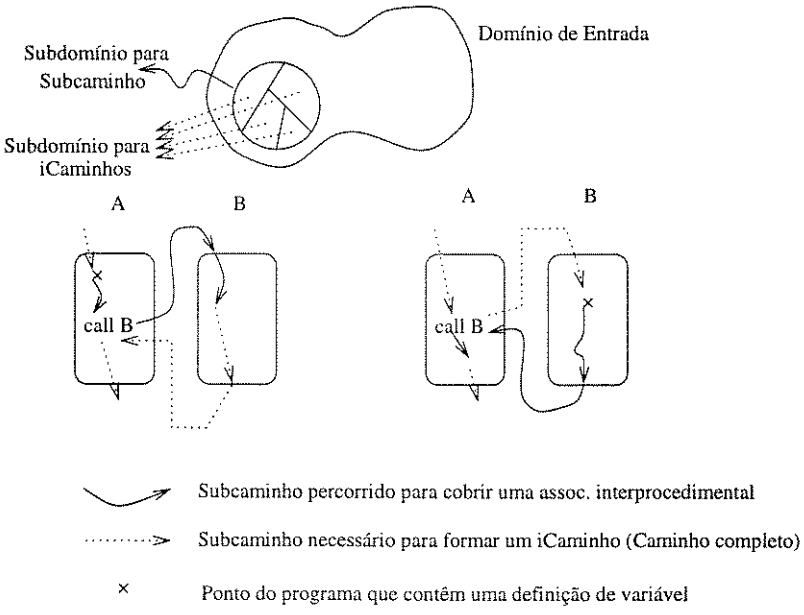

Figura 4.13: Subdomínios para Sub-caminhos e iCAMINHOS.

parcial entre critérios de teste, em especial com relação aos Critérios Baseados em Análise de Fluxo de Dados de Linnenkugel e Müllerburg [LM90], mesmo quando a habilidade de detecção de defeitos é considerada.

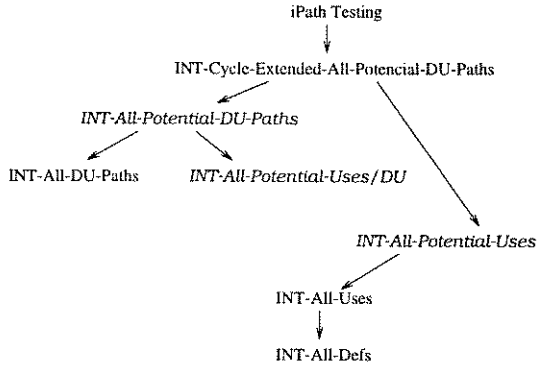

Figura 4.14: Habilidade de Detecção de Defeitos – Integração.

4.4 Considerações Finais

Neste capítulo foram introduzidas a Família de Critérios Potenciais Usos para o Teste de Integração e a correspondente família de critérios executáveis. Os critérios definidos, denominados Critérios Potenciais Usos de Integração, estão baseados no conceito de potencial uso. Assim como nos critérios definidos para o teste de unidade, uma característica que distingüe essa família de critérios das demais famílias de critérios de teste estrutural baseado em análise de fluxo de dados é que as associações de fluxo de dados são estabelecidas independentemente da ocorrência explícita de um uso de variável.

As principais características teóricas dos Critérios Potenciais Usos de Integração foram analisadas. Mostrou-se que os Critérios Potenciais Usos de Integração formam uma

hierarquia entre critérios de teste de integração baseados em análise de fluxo de dados e que nenhum outro critério de teste de integração baseado em análise de fluxo de dados inclui os Critérios Potenciais Usos de Integração. Outra característica importante dos Critérios Potenciais Usos de Integração é que eles, mesmo na presença de caminhos não executáveis, incluem o teste de ramos.

O principal resultado teórico apresentado diz respeito à habilidade de detecção de defeitos dos Critérios Potenciais Usos de Integração. Mostrou-se que esses critérios estabelecem uma hierarquia entre critérios de teste de integração baseados em análise de fluxo de dados e critério *Todos-iCaminhos*.

Capítulo 5

PokeInt - Aspectos de Implementação

Neste capítulo são apresentados a arquitetura e os principais aspectos relacionados à implementação de uma ferramenta de teste denominada – PokeInt (em referência à extensão da ferramenta POKE-TOOL [Cha91, Mal91] para o Teste de Integração) – que apóia a aplicação dos Critérios Potenciais Usos para Integração definidos no Capítulo 4.

A ferramenta PokeInt está baseada na reutilização de informação de teste gerada pela POKE-TOOL durante o teste de unidade, as informações do teste de unidade pertinentes à discussão neste capítulo são apresentadas no Apêndice A.2.

5.1 Reusando Informação do Teste de Unidade

Nesta seção apresenta-se a estratégia proposta para a implementação dos Critérios Potenciais Usos de Integração. Essa estratégia está apoiada no reuso de informação derivada durante o teste de unidade, para se obter os requisitos do teste de integração. O reuso de informação acontece através da combinação dos requisitos do teste de unidade – sub-caminhos ou associações interprocedimentais – para compor os requisitos do teste de integração. Essa estratégia aproveita o esforço aplicado na geração dos requisitos para o teste de unidade para se estabelecer os requisitos do teste de integração.

Os Critérios Potenciais Usos de Integração definidos no Capítulo 4 – Seção 4.1, consideram a integração dos módulos "dois-a-dois"; essa abordagem apóia as estratégias incrementais de integração, desde que o incremento seja de um par de módulos de cada vez. Foram definidos três critérios básicos: *INT-Todos-Potenciais-Usos*, *INT-Todos-Potenciais-Usos/DU* e, por fim, o *INT-Todos-Potenciais-DU-Caminhos.*

Tomando como exemplo o *Critério INT-Todos-Potenciais-Usos*, analisa-se como os elementos requeridos estabelecidos por esse critério podem ser representados. Considera-se a integração de dois módulos A e B, A o módulo chamador e B o chamado. A primeira parte do critério requer que para toda variável x de A – x pertencendo ao conjunto de variáveis de entrada presentes na chamada de A para B (considerando-se apenas uma chamada) – seja executado um sub-caminho a partir de toda definição de x até o ponto de chamada para o módulo B, e do ponto de entrada de B até todos os pontos onde exista um potencial uso de x (no módulo B x é representado pelo correspondente parâmetro formal). A segunda parte do critério requer que as definições de variáveis de saída internas ao módulo B sejam exercitadas até o nó de saída de B e de volta ao módulo A até todos os pontos onde exista um potencial uso.

Analisando-se a primeira parte do critério pode-se separar as associações requeridas

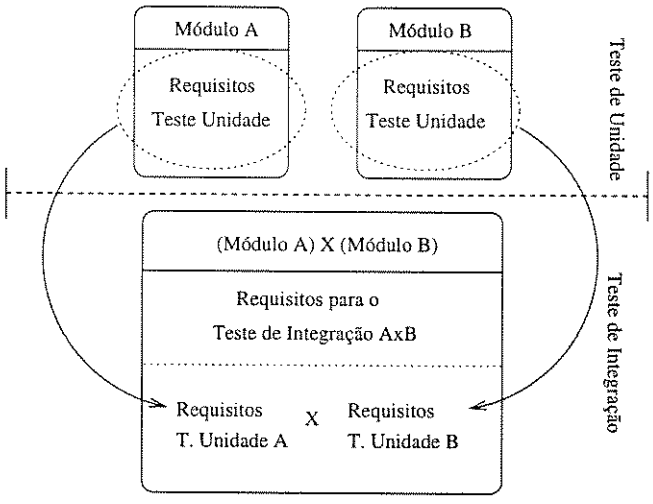

Figura 5.1: Requisitos de Teste de Integração

em duas partes, a primeira parte no módulo A e a segunda no módulo B; cada uma dessas partes é, na verdade, uma associação requerida pelo Critério Todos-Potenciais-Usos definido para o teste de unidade, só que agora devem ser exercitadas conjuntamente para satisfazer o critério de integração. A mesma análise é válida para a segunda parte dos critérios de integração. Dessa análise surge a idéia de se aproveitar os requisitos de teste identificados no teste de unidade para se implementar os Critérios de Teste de Integração, representados na Figura 5.1.

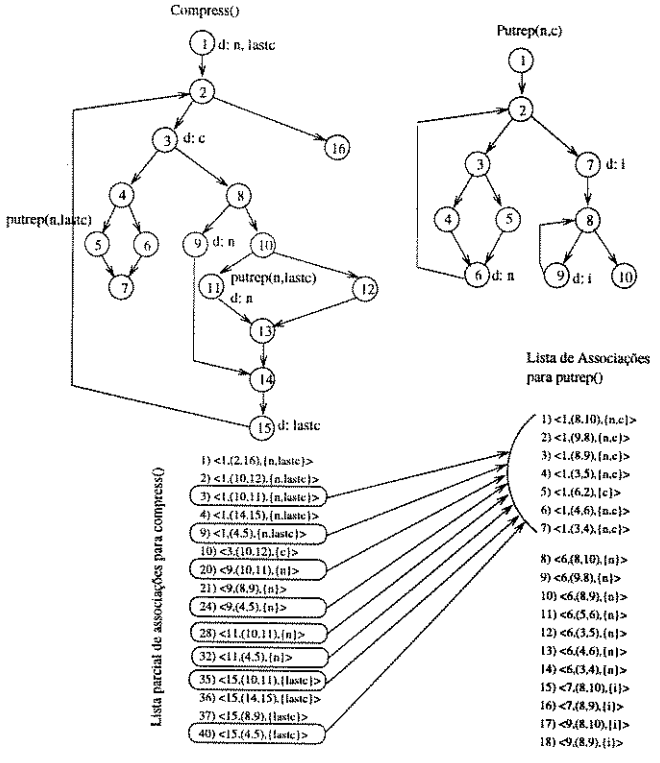

Figura 5.2: Selecionando Requisitos de Teste de Integração

Somente aqueles requisitos do teste de unidade que têm um significado interprocedimental são usados para compor requisitos de integração, como é o caso de requisitos envol-

vendo variáveis de comunicação e requisitos envolvendo nós de chamada a outros módulos. Observa-se no exemplo da Figura 5.2 dois módulos de um programa, o *compress* e o *putrep* [KP81]; o módulo compress chama o módulo putrep em dois pontos, no nó 5 e no nó 11. Das 40 associações identificadas pelo Critério Todos-Potenciais-Usos no módulo compress, apenas 8 envolvem o nó 5 ou o nó 11 e as variáveis n ou *lastc* (usadas como parâmetros reais, de entrada). Das 18 associações identificadas no módulo putrep apenas 7 envolvem o nó de entrada (nó 1) e as variáveis n ou c (parâmetros formais). Como nenhum dos parâmetros de putrep é parâmetro de saída, não vai existir nenhuma associação saindo de putrep e chegando em compress; portanto, a combinação das 8 associações de compress com as 7 associações de putrep, num total de 56 associações, formam a lista de elementos requeridos pelo Critério INT-Todos-Potenciais-Usos para o teste de integração dos módulos compress e putrep.

A Figura 5.3 mostra como os requisitos de teste de unidade são classificados e separados em conjuntos, para que possam ser combinados na definição dos requisitos de teste para o teste de integração. Os elementos requeridos do módulo chamador que atingem o ponto de chamada são alocados ao conjunto S_1 e aqueles que partem do ponto de chamada são alocados ao conjunto S_2. Os elementos requeridos do módulo chamador que partem do nó inicial ficam no conjunto S_3 e os que alcançam o nó final ficam no conjunto S_4. Os requisitos para o teste de integração são definidos pelas combinações $S_1 X S_3$ e $S_4 X S_2$. Essa estratégia pode ser usada tanto para se combinar associação quanto para se combinar sub-caminhos.

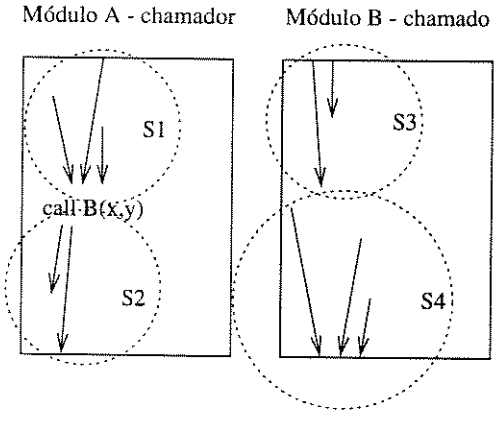

Figura 5.3: Associações com Relevância Interprocedimental

5.1.1 Outras Informações Reusadas

Os requisitos de teste derivados para o teste de unidade são apenas um tipo de informação que pode ser reusada no teste de integração. Na verdade, são identificados três tipos de reuso de informação no teste de integração:

1. Reusar requisitos de teste derivados para o teste de unidade ao se definir os requisitos do teste de integração;
2. Reusar casos de teste gerados para satisfazer requisitos do teste de integração, quando o software passa por uma manutenção e precisa ser retestado;
3. Reusar casos de teste gerados para o teste de unidade ao se aplicar o teste de integração.

O primeiro tipo foi discutido no início desta seção e constitui a base da implementação dos Critérios Potenciais Usos para o Teste de Integração. O segundo tipo é o chamado *teste de regressão no nível de integração* e é discutido como trabalho futuro no Capítulo 6.

O terceiro tipo envolve a reaplicação de casos de teste gerados para o teste de unidade. Esses casos de teste são reusados diretamente no teste de integração do programa. Esta idéia está baseada numa estratégia de teste apresentada por Pressman [Pre92]; segundo a estratégia apresentada, o teste do programa inicia no nível de módulo e evolui para o teste do programa como um todo; nesse sentido, o teste de integração só se inicia quando todos os módulos tiverem sido testados individualmente.

O ambiente para o teste de unidade contém o módulo que vai ser testado, um módulo responsável por passar os casos de teste para o módulo que vai ser testado e mostrar suas saídas – denominado "driver", e os simuladores dos módulos subordinados ao módulo que vai ser testado – denominados "stubs".

Considere-se o ambiente de teste de unidade, apresentado na Figura 5.4-B, para um programa com uma estrutura como a apresentada na Figura 5.4-C; observa-se que a cada módulo testado ter-se-á um correspondente conjunto de casos de teste (Figura 5.4-A). No caso do módulo chamar outros módulos subordinados, esses módulos são substituídos por "stubs" durante o teste de unidade; mas, de qualquer forma, os casos de teste gerados para executar o módulo em teste, eventualmente, executam as chamadas aos "stubs". No exemplo da Figura 5.4 o conjunto de casos de teste é subdividido em 4 subconjuntos; o subconjunto 1 executa a chamada ao "stub_b", o subconjunto 2 executa a chamada ao "stub_c", o subconjunto 3 executa a chamada ao "stub_d" e, finalmente, o subconjunto 4 não executa nenhuma chamada.

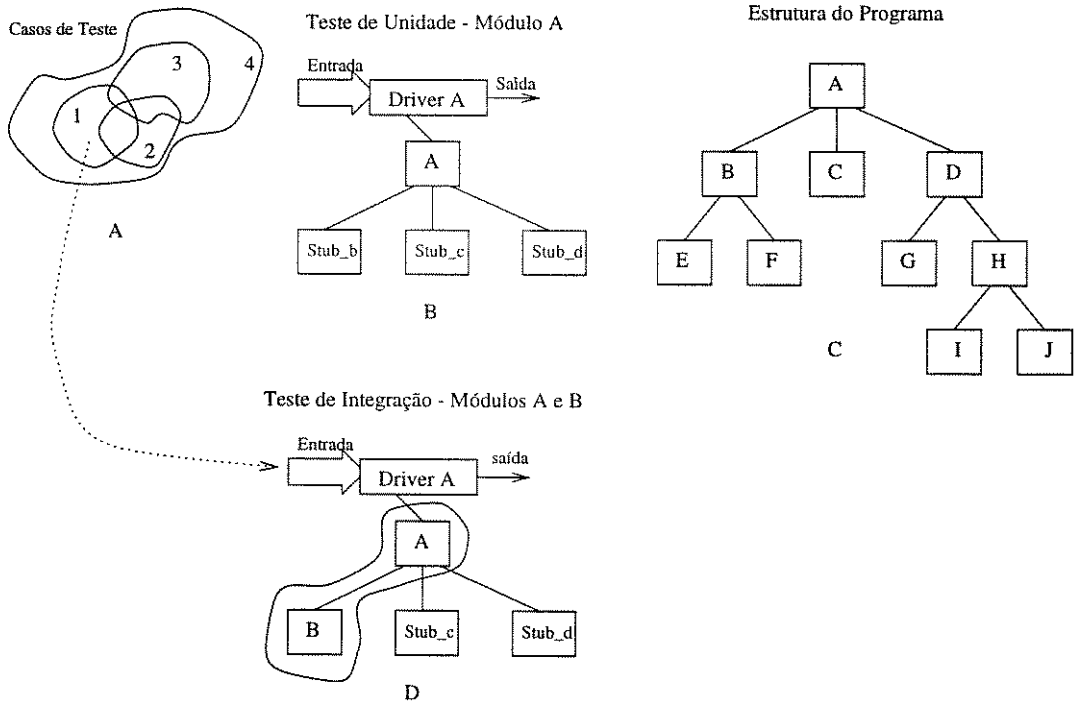

Figura 5.4: Seleção de Casos de Teste para o Teste de Integração

Ao se aplicar o teste de integração entre, por exemplo, os módulos A e B o subconjunto de casos de teste que, durante o teste de unidade, executavam a chamada ao "stub_b" pode ser reaplicado – subconjunto 1 no exemplo da Figura 5.4-A. Nesta situação os casos

de teste passam a executar a chamada ao módulo real ao invés de executar a chamada ao "stub".

Esse procedimento pode ser usado para se gerar um conjunto de casos de teste inicial para o teste de integração. Após avaliada a adequação desse conjunto inicial aos critérios de teste de integração, pode-se selecionar novos casos de teste com o objetivo de aumentar a cobertura obtida.

A implementação de um módulo para selecionar os casos de teste que podem ser aplicados no teste de integração é independente da implementação da ferramenta Poke-Int. Segue uma especificação inicial para esse módulo:

> **Módulo Reusando Casos de Teste de Unidade no Teste de Integração**
>
> O conjunto de casos de teste gerados para o teste de unidade de um programa deve ser percorrido com o objetivo de se recuperar aqueles casos de teste que tenham influência no teste de integração.
>
> Os casos de teste de interesse são aqueles que executam chamadas a "stubs" durante o teste de unidade.
>
> Deve-se fornecer o nome do módulo e quais são os "stubs" de interesse, a ferramenta separa os casos de teste que executam esses "stubs".

Um esboço inicial dos dois primeiros níveis do diagrama de fluxo de dados para esse módulo é apresentado na Figura 5.5.

5.2 Cálculo de Sinônimos

Um *sinônimo* ("alias") existe sempre que duas ou mais variáveis estão na verdade indicando uma mesma posição de memória; as formas mais comuns de sinônimos ocorrem com o uso de variáveis do tipo ponteiro e através de chamadas de procedimentos.

A informação de fluxo de dados interprocedimental é útil em muitas técnicas de teste e análise, incluindo teste baseado em análise de fluxo de dados, teste de regressão [LW90] e determinação de fatias de programa ("program slicing") [ADS93, Duc93, LH93]. Em programas com sinônimos, essas técnicas de teste e análise podem fornecer resultados inválidos ou insatisfatórios, a não ser que a análise de fluxo de dados considere o efeito causado pelos sinônimos [HR96].

Harrold e Rothermel [HR96] propõem um algoritmo para calcular o conjunto de sinônimos de um programa; a estratégia adotada separa o cálculo dessa informação módulo por módulo; quando os módulos são integrados a informação de sinônimos do módulo chamado é reusada e propagada para o módulo chamador. Essa abordagem está baseada na proposta por Pande, Landi e Ryder [PLR94], que propaga a informação de sinônimos para o programa como um todo, considerando sinônimos decorrentes do uso de variáveis do tipo ponteiro e chamadas de procedimentos.

No caso dos Critérios Potenciais Usos de Integração, apenas a informação de sinônimos referente às variáveis presentes em chamadas a procedimentos, como parâmetros tanto de entrada como de saída, deve ser determinada. O problema do tratamento de sinônimos decorrente do uso de ponteiro já foi resolvido através da abordagem conservadora [VMJ97b] apresentada no Capítulo 3; essa abordagem não pretende identificar com precisão as associações de fluxo de dados existentes em um módulo do programa; ao invés disso, pretende

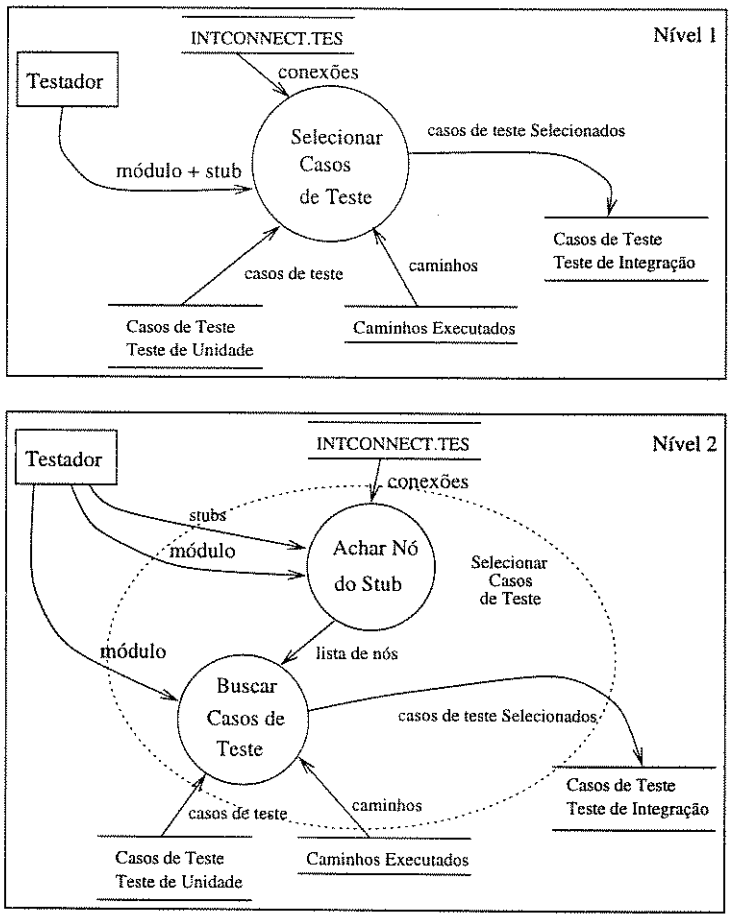

Figura 5.5: Diagrama de Fluxo de Dados – Seleciona Casos de Teste.

não deixar de exigir alguma associação que possa existir - daí dizer-se que se trata de uma abordagem conservadora.

Portanto, o problema que ainda precisa ser tratado é o da ocorrência de sinônimos devido às chamadas de procedimentos. Essas chamadas associam os parâmetros reais (que ocorrem no módulo chamador) a parâmetros formais (que ocorrem no módulo chamado); as variáveis envolvidas estão, na verdade, mapeando as mesmas posições de memória e, conseqüentemente, estão envolvidas nos mesmos fluxos de dados interprocedimentais.

A proposta é derivar tuplas (M, N, c, V_m, V_n) indicando as correspondências entre cada variável V_m do módulo M com a variável V_n do módulo N para cada chamada c de M para N. E, a partir daí, rastrear as associações interprocedimentais que envolvem as mesmas variáveis, independentemente do nome específico que elas assumam em cada módulo. As tuplas (M, N, c, V_m, V_n) são a realização do mapeamento x para $formal_c(x)$ conforme a terminologia apresentada no Capítulo 2 e usada na definição dos Critérios Potenciais Usos de Integração.

5.3 Modelos de Implementação - Integração

5.3.1 Modelo de Fluxo de Controle

Conforme apresentado no Capítulo 2 no modelo de fluxo de controle interprocedimental adotado, um programa P é representado por um multi-grafo direcionado $CG(P) = (\mathcal{N}, \mathcal{E}, s)$, onde módulos são associados a nós $n \in \mathcal{N}$ e os possíveis fluxos de controle entre módulos estão associados a arcos $e \in \mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$. Como CG é um multi-grafo, pode existir mais de um arco entre dois módulos. Qualquer nó no grafo pode ser alcançado a partir do nó s, chamado *nó raiz*. O grafo CG representando um programa P é chamado de *grafo de chamada*. Cada arco $e \in \mathcal{E}$ representa um par (M_i, N_i), $0 < i \leq k$, uma das k chamadas ao módulo N no módulo M. O modelo de fluxo de controle interprocedimental representando um programa através de um grafo de chamada também é utilizado por Linnenkugel e Müllerburg [LM90].

Um exemplo de grafo de chamada é apresentado na Figura 5.6. Nesse exemplo de representação os módulos que apresentam recursão dão origem a um arco saindo e voltando para o próprio módulo, como é o caso do módulo *amatch* na Figura 5.6.

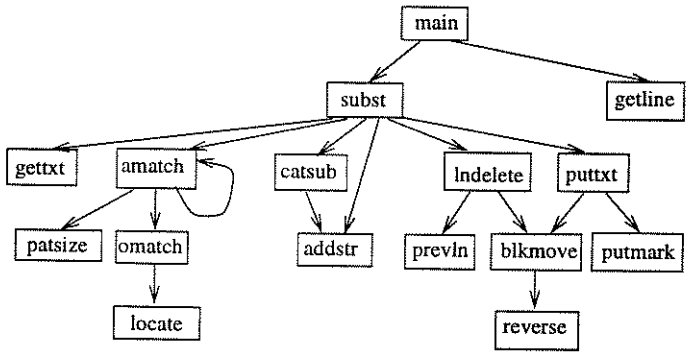

Figura 5.6: Grafo de Chamada - subst.c.

5.3.2 Modelo de Instrumentação

A instrumentação tem como objetivo possibilitar a análise posterior à execução dos casos de teste; por exemplo, a análise de adequação dos casos de teste executados aos elementos requeridos pelos critérios de teste. Para tanto a instrumentação deve alterar o programa em teste, através da inserção de código fonte, gerando uma nova versão do programa – denominado *programa instrumentado*. No caso do teste de unidade aplicado com o suporte da ferramenta POKE-TOOL, a unidade instrumentada é armazenada em um arquivo chamado *testeprog.c*; um exemplo de unidade instrumentada é apresentado no Apêndice A.2.

A instrumentação no nível de unidade consiste, basicamente, em inserir pontas de provas nos blocos de comandos correspondentes a cada nó do grafo de programa da unidade em teste. Desta forma, possibilita-se a identificação do caminho percorrido, dentro da unidade, pelo caso de teste fornecido. Uma ponta de prova nada mais é que um comando de escrita em arquivo; a cada bloco percorrido grava-se o número do nó correspondente, obtendo-se assim um "trace" do caso de teste fornecido.

Para permitir a análise para os critérios de integração após a execução do casos de teste, a instrumentação para o teste de unidade deve ser estendida. Deve-se incluir um comando de escrita que indique o módulo chamador e o módulo chamado para cada ocorrência de um comando de chamada no programa. Desta forma, pode-se obter um "trace" da execução interprocedimental de um caso de teste, permitindo a avaliação para os critérios de teste de integração.

Para o caso dos critérios "pairwise" pode-se incorporar um mecanismo que habilite a instrumentação apenas para o par de módulos que está sendo testado a cada momento, permitindo a obtenção de um rastreamento da execução mais significativo.

5.3.3 Modelo de Fluxo de Dados

Como já foi descrito no Capítulo 2, existem três tipos de ocorrências de variáveis em um programa: *definição*, *uso* e *indefinição*. No caso dos Critérios Potenciais Usos é suficiente associar a cada nó do grafo de programa apenas o conjunto de variáveis definidas no bloco de comandos correspondente, já que os critérios não consideram o uso explícito de uma variável para derivar os requisitos de teste. O mesmo acontece para os Critérios Potenciais Usos de Integração.

Além disso, apenas as variáveis globais e as que são parâmetros reais ou formais devem ser consideradas na análise de fluxo de dados interprocedimental. Essas variáveis são as responsáveis pelo fluxo de dados entre os módulos do programa. No caso do módulo chamador os parâmetros reais presentes em todas as chamadas ao módulo chamado devem ser considerados.

Os parâmetros presentes em uma chamada de procedimento podem ser classificados em dois tipos, *parâmetro de entrada* e *parâmetro de saída*. Os parâmetros de entrada não são redefinidos na chamada, enquanto que os parâmetros de saída podem ser redefinidos. A passagem de valores entre procedimentos através da passagem de parâmetros pode ser feita por: *valor*, *referência* ou *nome* [GJ87]. 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 chamadas *definições por referência*. Essa distinção é usada na determinação das associações de fluxo de dados para o teste de unidade [Mal91] e é fundamental na determinação dos elementos requeridos que envolvem variáveis do tipo ponteiro, como descrito no Capítulo 4. Da mesma forma,

essa distinção é fundamental na determinação dos elementos requeridos para o teste de integração segundo os Critérios Potenciais Usos de Integração; ou seja, ao se considerar uma definição por referência como sendo uma definição de variável – dando origem a um conjunto de associações de fluxo de dados – mas não uma redefinição de uma variável definida anteriormente, permite-se que se tenha uma abordagem conservadora na determinação das associações de fluxo de dados de um programa, tanto no teste de unidade quanto no teste de integração.

5.4 Arquitetura da Ferramenta PokeInt

A ferramenta PokeInt envolve a geração do grafo de chamada do programa, a identificação das conexões entre os módulos, a construção dos conjuntos S_1, S_2, S_3 e S_4 para cada conexão identificada (para cada critério de teste), a geração da lista de elementos requeridos combinados S_1XS_3 e S_4XS_2 e a avaliação dos casos de teste executados para determinar quais elementos requeridos foram exercitados.

A Figura 5.7 mostra os módulos que compõem a ferramenta PokeInt. O módulo *CallGraph* lê o código do programa traduzido para a linguagem intermediária e o programa fonte original para gerar o grafo de chamada do programa; o módulo *Connection* percorre cada conexão do grafo de chamada derivando os conjuntos de elementos requeridos e o módulo *Evaluation* identifica quais elementos requeridos foram efetivamente exercitados pelos casos de teste.

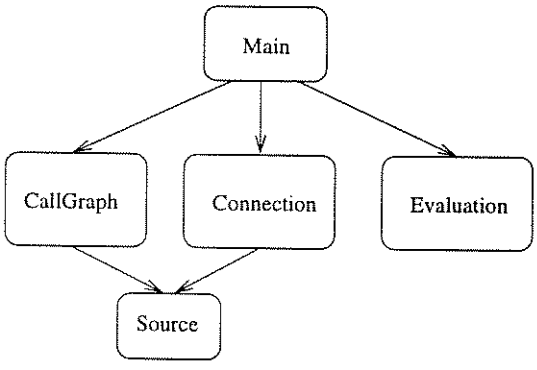

Figura 5.7: Principais Módulos da Ferramenta PokeInt

5.4.1 Grafo de Chamada

Para se obter o grafo de chamada, inicialmente uma versão do programa traduzida para uma linguagem intermediária – LI [1] – é percorrida para se identificar todos os comandos "@módulo", que indicam o início do código em LI correspondente àquele módulo. Essa fase inicial gera uma lista com todos os módulos do programa. Numa segunda etapa identificam-se as conexões de um módulo com outros módulos. Essa identificação é feita percorrendo-se cada módulo na LI até encontrar um comando "$S" ou "$C" que corresponda, no programa original, a uma chamada de procedimento. É nessa busca que se utilizam as informações de ligação entre a LI e o programa original. À medida que as chamadas a procedimento são identificadas vão sendo gerados os pares $(chamador, chamado)$ que representam os arcos do

[1] Um exemplo de programa traduzido para a LI é apresentado no Apêndice A.2.

grafo de chamada. Esses pares são gravados em um arquivo chamado *intconnec.tes*, como mostra a Figura 5.8. Segue o conteúdo do arquivo intconnect.tes do programa exemplo *compress*. As informações geradas pela POKE-TOOL para o teste de unidade do programa *compress* são apresentadas no Apêndice A.2.

```
compress putrep 5 448
compress putrep 11 605
main compress 1 1158
```

O arquivo intconnect.tes contém uma lista com o módulo chamador, o módulo chamado, o número do nó do grafo de programa onde ocorre a chamada e a posição no arquivo fonte original onde está a chamada.

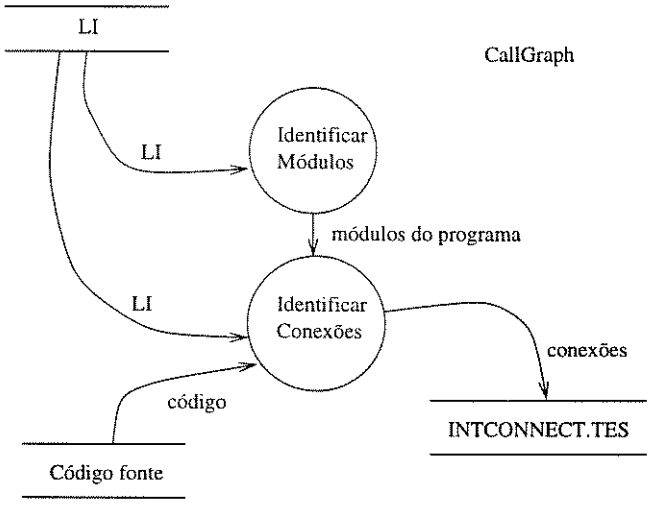

Figura 5.8: Gerando o Grafo de Chamada

5.4.2 Requisitos do Teste de Integração

Para se montar os conjuntos de associações usadas para compor os elementos requeridos para o teste de integração deve-se analisar as associações correspondentes a cada módulo do programa e filtrar aquelas com interesse interprocedimental; inicialmente identificam-se as variáveis envolvidas na interface entre os dois módulos (chamador e chamado), o nó de chamada e o nó de retorno (no módulo chamado). Com essa informação é possível separar e classificar as associações nos conjuntos S_1, S_2, S_3 e S_4. Isso é feito pelo módulo *Connection*, Figura 5.9.

A seguir encontra-se a lista de associações divididas em conjuntos para o programa compress.

```
Calling: compress
Called: putrep  5
Calling:
lastc
n
Called:
c
n
```

```
S1= 9 24 32 40
S2=
S3= 1 2 3 4 5 6 7
S4= 1 8 15 17

Calling: compress
Called: putrep  11
Calling:
lastc
n
Called:
c
n
S1= 3 20 28 35
S2= 25 26 27 28 29 30 31 32
S3= 1 2 3 4 5 6 7
S4= 1 8 15 17

Calling: main
Called: compress  1
Calling:
Called:
lastc
n
S1=
S2=
S3= 1 2 3 4 5 6 7 8 9
S4= 1 14 17 25 33
```

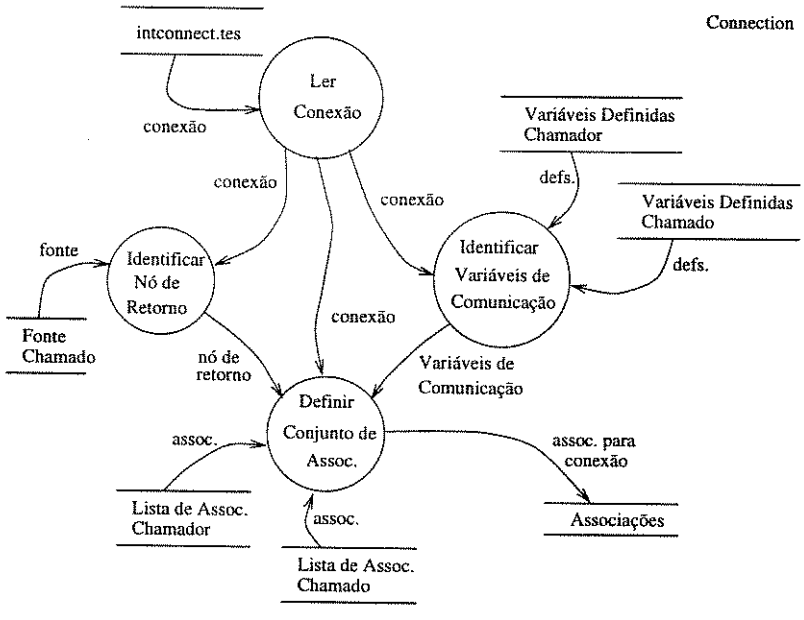

Figura 5.9: Gerando os Conjuntos de Associações

5.4.3 Avaliação para o Teste de Integração

A avaliação é uma função que verifica se o conjunto de caminhos executados satisfaz um dado critério de teste. Caso isso não ocorra deve ser fornecida uma lista dos elementos

requeridos ainda não exercitados. Para se fazer essa avaliação os elementos requeridos dos critérios são representados através de descritores, autômatos finitos, que devem ser satisfeitos para que a associação ou caminho seja considerado exercitado. Para exercitar um descritor é necessário que exista um sub-caminho, dentro de um caminho do programa, que realize as transições do *estado inicial* até o *estado final* do autômato.

Da mesma forma que os elementos requeridos para o Teste de Integração são uma composição de elementos requeridos para o Teste de Unidade, os descritores para o Teste de Integração são uma composição de descritores para o Teste de Unidade. Para compor dois descritores basta incluir um identificador no estado que indique a transferência de controle de um módulo para outro, através de um comando de chamada.

Observe a seguir a composição de dois descritores para o programa compress. Considera-se a composição da associação $< 1,(10,11),\{n,lastc\} >$ do módulo compress com a associação $< 1,(8,10),\{n,c\} >$ do módulo putrep; os descritores correspondentes são: $N*\ 1\ N_{nv}*\ 10\ [N_{nv}*\ 10]*\ 11\ (N_{nv} = 2\ 3\ 4\ 5\ 6\ 7\ 8\ 10\ 12\ 13\ 14\ 16)$ e $N*\ 1\ N_{nv}*\ 8\ [N_{nv}*\ 8]*\ 10\ (N_{nv} = 2\ 3\ 4\ 5\ 7\ 8\ 9\ 10)$, respectivamente.

$$N_{compress} * 1\ N_{nv_{compress}} * 10\ [N_{nv_{compress}} * 10] * 11\ N_{putrep} * 1\ N_{nv_{putrep}} * 8\ [N_{nv_{putrpe}} * 8] * 10$$

$$N_{nv_{compress}} = 2\ 3\ 4\ 5\ 6\ 7\ 8\ 10\ 12\ 13\ 14\ 16$$

$$N_{nv_{putrep}} = 2\ 3\ 4\ 5\ 7\ 8\ 9\ 10$$

$N_{unidade}$ é o conjunto de todos os nós daquela unidade. $N_{nv_{unidade}}$ é o conjunto de nós do $grafo(i)$ [2], com exceção daqueles nós onde ocorre definição da variável v.

Portanto, é necessário um caso de teste cujo caminho executado conduza à máquina de estados descrita acima ao seu estado final, ou seja, a mesma execução deve exercitar uma associação no módulo compress e uma associação no módulo putrep através da chamada à putrep no nó 11 de compress.

5.5 Considerações Finais

A estratégia de implementação definida para a implementação da ferramente Poke-Int está baseada no reuso de informação gerada durante a aplicação do teste de unidade; em especial, nas informações geradas pela ferramenta POKE-TOOL durante o teste de unidade. Essa estratégia pretende minimizar tanto o esforço para a implementação da ferramenta Poke-Int quanto o esforço para a aplicação do teste de integração.

[2] $Grafo(i)$ [MCJ88, Mal91] é o grafo que contém a partir do nó i todos os sub-caminhos alcançáveis sem redefinição das variáveis definidas em i.

Capítulo 6

Conclusões

Na primeira seção deste capítulo é apresentada uma síntese do trabalho desenvolvido; em seguida, são listadas as principais contribuições; finalmente, são discutidos alguns trabalhos de pesquisa futuros relacionados ao tema da tese.

6.1 Síntese do Trabalho

Os critérios de teste de software propostos nos últimos anos são, na sua maioria, critérios para o teste de unidade. Dessa forma, tanto critérios estruturais quanto baseados em erros têm sido definidos para se testar cada unidade isoladamente. Esses critérios não se adequam diretamente ao teste de integração, dado que levariam à definição de um número elevado de elementos requeridos. Para o teste de integração eram usados apenas critérios funcionais. Apesar de se saber que aproximadamente 40% dos defeitos presentes em produtos de software liberados são defeitos de integração, apenas recentemente têm surgido trabalhos que propõem o uso de critérios estruturais e baseados em erros para o teste de integração.

Este trabalho investigou, essencialmente, o uso de informações de fluxo de dados para derivar critérios de teste para o teste de integração de programas. Esse é um enfoque promissor já que a interface entre dois módulos é definida pelas variáveis utilizadas em sua interação.

Foram introduzidas a Família de Critérios Potenciais Usos de Integração e a correspondente Família de Critérios Potenciais Usos de Integração Executáveis. Os Critérios Potenciais Usos de Integração, assim como os critérios definidos para o teste de unidade, estão baseados no conceito de potencial-uso; desta forma, requisitos de teste são estabelecidos sem que usos de variáveis sejam necessariamente identificados. Essa pequena mas fundamental diferença garante alguns dos resultados identificados na análise teórica dos critérios.

Estabeleceu-se uma terminologia única para a definição dos critérios de teste, como para o teste de unidade quanto para o teste de integração. Os critérios apresentados na literatura foram reescritos segundo a terminologia para facilitar a comparação entre eles.

Um aspecto importante para a comparação entre critérios de teste é a habilidade de detecção de defeitos. Define-se um relacionamento R tal que, se $R(C_1, C_2)$, então um conjunto de casos de teste que satisfaz C_1 para um programa P tem mais chance de revelar a presença de defeitos que um conjunto de casos de teste que satisfaz C_2 para o mesmo programa P.

Os Critérios Potenciais Usos foram comparados aos Critérios de Rapps e Weyuker

segundo sua habilidade de detectar defeitos; para tanto utilizou-se a relação *cobre propriamente*, definida por Frankl e Weyuker [FW93a]. Identificou-se que os Critérios Potenciais Usos estabelecem uma hierarquia de critérios entre o teste de ramos e o teste de caminhos e que nenhum outro critério de teste baseado em análise de fluxo de dados cobre os Critérios Potenciais Usos; esse resultado é análogo ao obtido por Maldonado [Mal91] na análise dos Critérios Potenciais Usos segundo a relação de inclusão.

Ainda no nível de unidade, foi utilizada uma abordagem para o teste de programas com uso de variáveis do tipo ponteiro seguindo a mesma estratégia do tratamento de vetores proposto por Maldonado [Mal91]. O uso desse tipo de variável provoca uma indeterminação no cálculo das associações de fluxo de dados, já que não se pode determinar estaticamente para qual posição de memória um ponteiro está apontando num certo ponto do programa. A abordagem proposta não pretende analisar individualmente cada posição de memória para a qual um ponteiro possa apontar; trata-as como uma única variável. Essa estratégia combinada com o modelo de fluxo de dados definido para os Critérios Potenciais Usos permitiu a definição de uma abordagem conservadora para a definição das associações de fluxo de dados que envolvam variáveis do tipo ponteiro.

Esses estudos, no nível do teste de unidade, motivaram a definição dos Critérios Potenciais Usos de Integração. As propriedades teóricas dos Critérios Potenciais Usos de Integração analisadas foram a análise de inclusão e a habilidade de detecção de defeitos. Tanto em relação à análise de inclusão como à habilidade de detecção de defeito demonstrou-se que os Critérios Potenciais Usos de Integração mantêm uma hierarquia entre o teste de ramos e o teste de caminhos, característica que não está presente em nenhuma outra família de critérios de integração analisada. Além disso, demostrou-se que os Critérios Potenciais Usos de Integração mantêm uma hierarquia entre o teste de ramos e o teste de caminhos mesmo na presença de caminhos não executáveis.

O primeiro passo para a condução de experimentos que permitam avaliar o custo, a força e a eficácia dos critérios definidos é a construção de uma ferramenta de apoio à aplicação desses critérios. Nesse sentido foram analisados os aspectos essenciais para a implementação da ferramenta denominada *Poke-Int*, que suporte a aplicação dos Critérios Potenciais Usos de Integração no teste estrutural de programas.

Um dos trabalhos futuros desta tese é a implementação da ferramenta Poke-Int e a condução de experimentos para a avaliação empírica dos critérios. Na seção seguinte são apresentadas as principais contribuições desta tese e, em seguida, outros trabalhos de pesquisa relacionados à tese.

6.2 Contribuições da Tese

Segue uma lista das principais contribuições desta tese:

- Definição dos Critérios Potenciais Usos de Integração e análise de suas propriedades teóricas. Os Critérios Potenciais Usos de Integração são definidos para o teste de integração dois-a-dois entre módulos de um programa; essa abordagem pode ser utilizada com estratégias incrementais de integração. Os Critérios Potenciais Usos de Integração estão baseados no conceito de potencial-uso; portanto, não requerem a ocorrência explícita de um uso de variável para estabelecer uma associação interprocedimental de fluxo de dados.

- Definição das diretrizes básicas de implementação da ferramenta Poke-Int, que deverá implementar os Critérios Potenciais Usos de Integração. Essas diretrizes básicas incluem a estratégia de implementação baseada em reuso de informação. Segundo essa estratégia os requisitos do teste de integração são derivados a partir de uma concatenação de requisitos para o teste de unidade.
- Análise da habilidade de detecção de defeitos dos Critérios Potenciais Usos. Essa análise é uma complementação dos estudos teóricos, apresentados por Maldonado [Mal91], para esses critérios. A análise da habilidade de detecção dos Critérios Potenciais Usos, ainda no nível de unidade, motivou o estudo e a definição dos critérios para o teste de integração; pôde-se comprovar que os Critérios Potenciais Usos mantêm uma hierarquia de critérios entre o teste de ramos e o teste de caminhos mesmo quando a habilidade de detectar defeitos é considerada. Outro fato importante é que nenhum outro critério de teste estrutural tem maior probabilidade de detectar defeitos em um programa que os Critérios Potenciais Usos. Essas propriedades são preservadas pelos Critérios Potenciais Usos de Integração.
- Definição de uma abordagem conservadora para o teste de programas com uso extensivo de variáveis do tipo ponteiro. Essa abordagem foi comparada com as abordagens de Ostrand e Weyuker [OW91] e de Horgan e London [HL91], comprovando-se ser mais exigente que essas abordagens.

6.3 Trabalhos Futuros

6.3.1 Confiabilidade de Software

O crescimento das aplicações de sistemas baseados em computador nos últimos anos, em especial, em aplicações críticas, tem motivado o estudo e a caracterização de modelos de estimativa da confiabilidade desses sistemas, e conseqüentemente, de modelos de confiabilidade da parte implementada em software.

O estudo da confiabilidade de software é a "ciência aplicada para prever, medir, e gerenciar a confiabilidade de software com a intenção de maximizar a satisfação do usuário". Esse ambiente de trabalho requer modelos de confiabilidade que auxiliem os engenheiros de confiabilidade de software a prever e controlar o processo de desenvolvimento do software de forma que o produto final satisfaça os critérios de confiabilidade.

A confiabilidade de um programa é definida como a probabilidade de o programa não falhar em um dado ambiente durante um dado intervalo de tempo [MIO87]. Essa é uma métrica importante para se decidir sobre a entrega (ou comercialização) do software e também pode ser usada para prever a confiabilidade de um software já em operação.

A modelagem da confiabilidade do software consiste em fazer uma descrição probabilística precisa da confiabilidade, baseada nas suposições sobre os fatores que afetam a confiabilidade, e nos resultados obtidos dos dados experimentais. Os modelos são utilizados para se medir a confiabilidade, analisar dados de falha, fazer inferências sobre o comportamento futuro do software e para auxiliar o processo de tomada de decisões durante o teste e a depuração do software.

Grande parte da literatura sobre modelos de confiabilidade de software está concentrada na descrição da evolução da confiabilidade durante a fase de teste do software. Nesse sentido, até então, todos os modelos de confiabilidade de software utilizam dados sobre falhas obtidos através da aplicação do teste funcional, com a suposição de que a confiabilidade

cresce automaticamente com o progresso do teste [Ham92]. Como conseqüência, um modelo pode se enquadrar bem para um software mas não ser adequado para outro. Iannino, et al. [IMOL84] propuseram um conjunto de critérios de comparação entre modelos de confiabilidade de software com o objetivo de estabelecer uma base comum de comparação entre os diversos modelos.

Chen, et al. [CMR95] propõem a definição de modelos de confiabilidade de software que levem em consideração a cobertura obtida pelos casos de teste segundo critérios estruturais de teste. Esta idéia está baseada na observação de que novos defeitos no software são descobertos à medida que novas partes do software são exercitadas pelos casos de teste; isto é, à medida que se aumenta a cobertura. Crespo [Cre97] propôs modelos de confiabilidade baseados na cobertura dos Critérios Potenciais Usos no nível do teste de unidade. Uma proposta é estender os resultados obtidos, considerando os Critérios Potenciais Usos para Integração, com o objetivo de investigar a relação entre a cobertura desses critérios com a confiabilidade do software.

6.3.2 Teste de Programas Orientados a Objetos

A programação orientada a objetos é, atualmente, muito usada em aplicações de sistemas baseados em computador. Uma das características da orientação a objetos é permitir que o projetista produza componentes reusáveis de software [Mey87]. Esses componentes, segundo seu defensores, reduzem o custo de produção do software já que uma vez desenvolvidos os componentes eles podem ser reusados em futuras versões. Os aspectos relativos à reusabilidade do software vão muito além dessa discussão [BP89b, BP89a, GHJV94]; de qualquer forma, a orientação a objetos representa, com certeza, uma alternativa para se atingir um nível alto de produtividade e qualidade no desenvolvimento de software.

Entretanto, muita pesquisa e experiência prática com a orientação a objetos tem se concentrado nas áreas de projeto e programação [CAB+94, Jor90, KM90, Rin96]; poucos trabalhos têm sido publicados examinando os efeitos das técnicas de orientação a objetos nos outros estágios do ciclo de vida do software. O fato de nos últimos anos terem aparecido uma série de artigos sobre teste de programas orientados a objetos [SR92, PBC93, MK94, Bin94, HR94, KGH+95, OI95, Bin96, TD97, CSW97] motiva a extensão do trabalho apresentado nesta tese para o teste de programas orientados a objetos, em especial, para o teste no nível de classes.

O que se propõe como extensão é a análise da aplicação dos Critérios Potenciais Usos de Integração para o teste de programas orientados a objetos, e, em especial, para o teste no nível de classes.

6.3.3 Teste de Regressão no Nível de Integração

Sistemas baseados em computador e suas aplicações evoluem à medida que se adaptam às mudanças no ambiente, às mudanças de requisitos, aos novos conceitos e às novas tecnologias. O software cresce no número de funções, componentes e interfaces. Módulos antigos podem ser expandidos para usos além dos definidos no projeto original. Desta forma, modificações no software são inevitáveis. Essa problemática já se refletiu na necessidade de se identificar e gerenciar a configuração do software à medida que ele é modificado [Ber84] e também deve se refletir na estratégia adotada para retestar o software depois de uma modificação. Essa necessidade é justificável já que a maior parte do tempo gasto na manutenção do software concentra-se em modificá-lo e retestá-lo. Portanto, um *teste de regressão* eficiente

e efetivo pode reduzir os custos da manutenção.

Teste de regressão é um processo de teste aplicado depois que um programa é modificado. Envolve testar o programa modificado com alguns casos de teste para reestabelecer a confiança de que o programa vai executar de acordo com a especificação (que também pode ter sido modificada). Retestar o programa com todos os casos de teste existentes tem geralmente um custo muito alto devido ao tamanho do programa e à freqüência dos testes, mas essa é uma prática comum. O teste de regressão pode se tornar mais eficiente com estratégias que enfatizam o reteste parcial.

Nesse sentido, Granja [Gra97] desenvolveu uma ferramenta de apoio ao teste de regressão – **RePoke-Tool** ("Regression Testing support for Potential-Uses Criteria Tool"), para ser aplicada em unidades que foram originalmente testadas usando-se a ferramenta de teste POKE-TOOL. Esse trabalho considera o teste de regressão no nível de unidade. Uma proposta é estender os resultados obtidos para se definir e comparar [RH96] uma estratégia para o *teste de regressão no nível de integração* [LW90] apoiada nos Critérios Potenciais Usos para Integração.

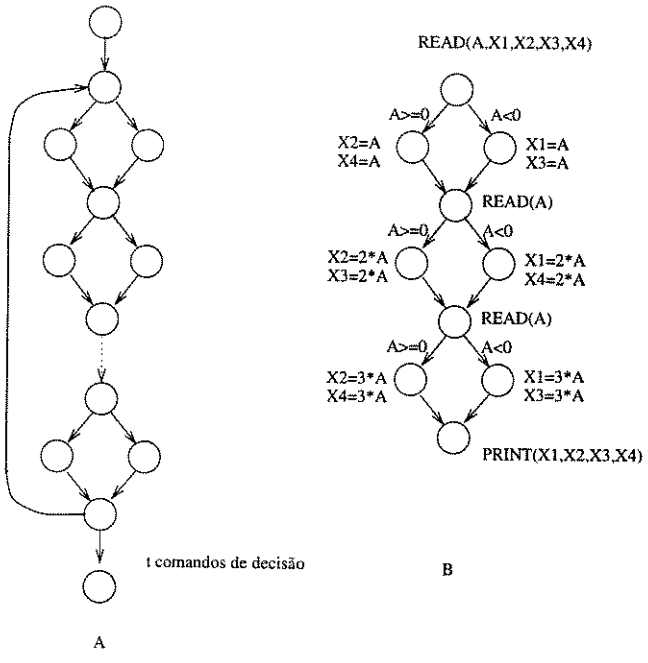

Figura 6.1: (A) Estrutura de Controle que Maximiza o Número de Potenciais DU-Caminhos; (B) Exemplo para a Análise de Complexidade.

6.3.4 Questões Relativas ao Custo

O custo de aplicação de um critério de teste pode ser estabelecido através de uma análise teórica ou através de uma análise empírica.

Sob o ponto de vista teórico o custo de um critério tem sido determinado pelo estudo da complexidade do critério. A complexidade de critérios estruturais tem sido determinada através da estimativa do número máximo de casos de teste requerido pelo critério no pior caso. Observa-se que para a determinação da complexidade de um critério de integração deve-se considerar qualquer grafo de fluxo de controle para os módulos integrados e qualquer distribuição de ocorrência de variáveis.

O primeiro passo na determinação da complexidade de um critério de teste estrutural consiste em determinar uma estrutura de fluxo de controle que maximize o número de elementos requeridos, considerando-se um fluxo de dados qualquer. Maldonado [Mal91] mostrou que o grafo de fluxo de controle da Figura 6.1-A 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 – t é o número de comandos de decisão do módulo. Sabe-se da análise de inclusão que o Critério Todos-Potenciais-DU-Caminhos inclui os critérios Todos-Potenciais-Usos e Todos-Potenciais-Usos/DU, assim sendo, 2^t é um limitante superior para a complexidade desses critérios. Segundo mostrado por Maldonado com o exemplo da Figura 6.1-B, verifica-se que 2^t casos de teste são requeridos e que a complexidade dos critérios Todos-Potenciais-Usos e Todos-Potenciais-Usos/DU é 2^t.

Uma proposta de trabalho futuro é estender a análise de complexidade para o caso de critérios de integração dois-a-dois, para tanto deve-se inicialmente determinar-se a estrutura de controle que maximiza o número de potenciais associações interprocedimentais.

A análise teórica da complexidade dos Critérios Potenciais Usos de Integração deve também ser complementada pela análise empírica do custo de aplicação dos critérios.

Referências Bibliográficas

[ADS93] H. Agrawal, R.A. DeMillo, and E.H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice & Experience*, 23(6):589–616, Junho de 1993.

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New Work, 1990.

[Ber84] E.H. Bersoff. Elements of software configuration management. *IEEE Trans. Soft. Eng.*, SE-10(1):79–87, Janeiro de 1984.

[Bin94] R.V. Binder. Design for – in object-oriented systems. *Communications of the ACM*, 27(9):87–101, Setembro de 1994.

[Bin96] R.V. Binder. Testing object-oriented software: A survey. *Software Testing, Verification and Reliability*, 6(3-4), Setembro de-Dezembro de 1996.

[BP89a] T.J. Biggerstaff and A.J. Perlis. *Software Reusability: Applications and Experience*, volume II. ACM Press, 1989.

[BP89b] T.J. Biggerstaff and A.J. Perlis. *Software Reusability: Concepts and Models*, volume I. ACM Press, 1989.

[BP94] V.R. Basili and R.T. Perricone. Software errors and complexity: An empirical study. *Communications of the ACM*, 27(1):42–52, Janeiro de 1994.

[BS89] J. Bieman and J. Schultz. Estimating the number of test cases required to satisfy the all-du-paths testing criteria. In *ACM SIGSOFT'89 – Third Symposium on Software Testing, Analysis and Verification*, Flórida, USA, Dezembro de 1989.

[Bud81] T.A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Computer Program Testing*, pages 129–148, Amsterdan: North Holland, Julho de 1981.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development – The Fusion Method.* Prentice Hall, 1994.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural dataflow analysis. In *SIGPlan Conf. Programming Language Design and Implementation*, pages 47–56, Atlanta, Georgia, Junho de 1988. ACM.

[Cha91] M.L. Chaim. Poke-tool — uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas – SP, Brasil, Abril de 1991.

[CHR82] L.A. Clarke, J. Hassel, and D.J. Richardson. A close look at domain testing. *IEEE Trans. Soft. Eng.*, 1982.

[Chu87] T. Chusho. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Trans. Soft. Eng.*, SE-13(5):509–517, Maio de 1987.

[CMR95] M. Chen, A.P. Mathur, and V. Rego. Effect of testing technique on software reliability estimates obtained using a time-domain model. *IEEE Trans. on Reliability*, 44(1):97–103, Março de 1995.

[CPRZ85] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil. A comparison of data flow path selection criteria. In *8th Int'l Conf. on Software Engineering*, pages 244–251, Agosto de 1985.

[CPRZ89] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Soft. Eng.*, 15(11):1318–1332, Novembro de 1989.

[Cre97] A.N. Crespo. Modelos de confiabilidade de software baseados em cobertura de critérios estruturais de teste. Tese de Doutorado, DCA/FEEC/UNICAMP, Campinas – SP, Brasil, Dezembro de 1997.

[CSW97] C-M Chung, T.K. Shih, and C.C. Wang. Object-oriented software testing and metric in z specification. *Information Sciences*, 98(1-4):175–202, Maio de 1997.

[Del97] M.E. Delamaro. Mutação de interface: Um critério de adequação interprocedimental para o teste de mutação. Tese de Doutorado, IFSC - USP, São Carlos – SP, Brasil, Junho de 1997.

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), Abril de 1978.

[DMM96] M.E. Delamaro, J.C. Maldonado, and A.P. Mathur. Integration testing using interface mutation. In *7th Int. Symp. Software Reliability Engineering*, pages 112–121, White Plains, NY, Outubro de 1996.

[Duc93] M. Ducassé. A pragmatic survey of automated debugging. In *First International Workshop on Automated and Algorithmic Debugging*, Maio de 1993.

[Fra87] F.G. Frankl. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD thesis, New York Univ, New York, Outubro de 1987.

[FW85] P.G. Frankl and E.J. Weyuker. A data flow testing tool. In *IEEE Softfair II*, Dezembro de 1985.

[FW86] P.G. Frankl and E.J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Workshop on Software Testing*, pages 4–13, Banff, Canada, Julho de 1986.

[FW88] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Soft. Eng.*, SE-14, Outubro de 1988.

[FW93a] P.G. Frankl and E.J. Weyuker. An analytical comparison of the fault-detecting ability of data flow testing techniques. In *15th International Conference on Software Engineering*, pages 415–424, Maio de 1993.

[FW93b] P.G. Frankl and E.J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. Soft. Eng.*, 19(3):202–213, Março de 1993.

[FW93c] P.G. Frankl and E.J. Weyuker. Provable improvements on branch testing. *IEEE Trans. Soft. Eng.*, 19(10):962–975, Outubro de 1993.

[FWH97] P.G. Frankl, S.N. Weiss, and C. Hu. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, Setembro de 1997.

[GG75] J. Goodenough and S.L Gerhart. Toward a theory of test data selection. *IEEE Trans. Soft. Eng.*, SE-1, 1975.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters.* Addison Wesley, 1994.

[GJ87] C. Ghessi and M. Jazayeri. *Programing Languages Concepts.* John Wiley and Sons, New York, 2nd edition, 1987.

[Gra97] I. Granja. Uma ferramenta de apoio ao teste de regressão. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas - SP, Brasil, Dezembro de 1997.

[Ham92] D. Hamlet. Are we testing for true reliability. *IEEE Software*, 9(4), Julho de 1992.

[Her76] P.M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3), Novembro de 1976.

[HGN93] D. Hamlet, B. Gifford, and B. Nikolik. Exploring dataflow testing of arrays. In *15th International Conference on Software Engineering*, pages 118–129, Baltimore, Maryland, Maio de 1993.

[HL91] J.R. Horgan and S. London. Data flow coverage and the c language. In *Fourth Symp. Soft. Testing, Analysis, and Verification*, pages 87–97, Outubro de 1991.

[How75] W.E. Howden. Methodology for the generation of program test data. *IEEE Trans. on Computer*, C-24(5):554–559, 1975.

[How78] W.E. Howden. An evaluation of the effectiveness of symbolic testing. *Software Practice & Experience*, 8, 1978.

[How87] W.E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, USA, 1987.

[HR94] M.J. Harrold and G. Rothermel. Performing data flow testing on classes. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, Dezembro de 1994.

[HR96] M.J. Harrold and G. Rothermel. Separate computation of alias information for reuse. In *1996 International Symposium On Software Testing and Analysis*, pages 107–120, Fevereiro de 1996. IEEE TSE, V.22, N.7, Julho de 1996.

[HS91] M.J. Harrold and M.L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, Março de 1991.

[Hua75] J.C. Huang. An approach to program testing. *Computing Surveys*, 7(3):113–128, Setembro de 1975.

[IMOL84] A. Iannino, J.D. Musa, K. Okumoto, and B. Littlewood. Criteria for software reliability model comparisons. *IEEE Trans. Soft. Eng.*, SE-10(6):687–691, Novembro de 1984.

[Jor90] D. Jordan. Implementation benefits of c++ language mechanisms. *Communications of the ACM*, 33(9):61–64, Setembro de 1990.

[KGH+95] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y-S Kin, and Y-K Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–87, Outubro de 1995.

[Kin76] J.C. King. Symbolic execution and program testing. *Comm. of ACM*, 19(7):385–394, 1976.

[KL85] B. Korel and J. Laski. A tool for data flow oriented program testing. In *Softfair II*, pages 34–38, San Francisco, CA, Dezembro de 1985.

[KM90] T. Korson and J.D. McGregor. Understanding object-oriented: A unifying pradigm. *Communications of the ACM*, 33(9):61–64, Setembro de 1990.

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

[LH93] L.D. Larsen and M.J. Harrold. Slicing object-oriented software. Technical report, Dept. of Computer Science, Clemson University, Clemson, SC 29634-1906, 1993.

[LK83] J.W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Soft. Eng.*, SE-9(3), Maio de 1983.

[LM90] U. Linnenkugel and M. Müllerburg. Test data selection criteria for (software) integration testing. In *First International Conference on Systems Integration*, pages 709–717, Morristown, New Jersey, Abril de 1990.

[LR90] W. Landi and B.G. Ryder. Pointer-induced aliasing: A problem classification. In *17th Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Janeiro de 1990.

[LR92] W. Landi and B.G. Ryder. A safe aproximate algorithm for interprocedural pointer aliasing. In *SIGPLAN92 Conference on Programming Language Design and Implementation*, pages 235–248, Junho de 1992.

[LW90] H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *1990 Conference on Software Maintenance*, pages 290–301, San Diego, CA, Novembro de 1990.

[Mal91] J.C. Maldonado. Critérios potenciais usos: Uma contribuição ao teste estrutural de software. Tese de Doutorado, DCA-FEE-UNICAMP, Campinas - SP, Brasil, Julho de 1991.

[MCJ88] J.C. Maldonado, M.L. Chaim, and M. Jino. Resultados do estudo de uma família de critérios de teste de programas baseada em fluxo de dados. Technical Report RT/DCA-001/88, DCA-FEEC-UNICAMP, Campinas, SP - Brazil, 1988.

[MCJ89] J.C. Maldonado, M.L. Chaim, and M. Jino. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos. In *XXII Congresso Nacional de Informática*, São Paulo, SP, Brazil, Setembro de 1989.

[Mey87] B. Meyer. Re-usability: The case for object-oriented design. *IEEE Software*, 4(2):50–63, 1987.

[MF96] D.I.S. Marx and P.G. Frankl. The path-wise approach to data flow testing with pointer variables. In *ISSTA '96*, Fevereiro de 1996.

[MIO87] J.D. Musa, A. Ianino, and K. Okumoto. *Software Reliability - Mesurement, Prediction, Application.* McGraw-Hill, New York, 1987.

[MK94] J.D. McGregor and T.D. Korson. Integrated object-oriented and development process. *Communications of the ACM*, 37(9):59–86, Setembro de 1994.

[MW93] A.P. Mathur and W.E. Wong. Evaluation of the cost of alternate mutation testing strategies. In *7th Brazilian Symp. on Software Engineering*, Rio de Janeiro - Brasil, Outubro de 1993.

[MW94a] A.P. Mathur and W.E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *J. of Software Testing Verification and Reliability*, 4(1):9–31, Março de 1994.

[MW94b] A.P. Mathur and W.E. Wong. A theoretical comparison between mutation testing and data flow based on test adequacy criteria. In *1994 ACM Computer Science Conference*, Phoenix - Arizona, Março de 1994.

[MW95] A.P. Mathur and W.E. Wong. Fault detection effectiveness of mutation and data flow testing. *Software Quality J.*, 4(1):69–83, Março de 1995.

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

[Nta84] S.C. Ntafos. On required element testing. *IEEE Trans. Soft. Eng.*, SE-10(6), Novembro de 1984.

[Nta88] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Soft. Eng.*, 14(6):868–873, Junho de 1988.

[OI95] A.J. Offutt and A. Irvine. Testing object-oriented software using the category-partition method. In *17th Int. Conf. on Technology of Object-Oriented Languages and Systems TOOLS*, pages 293–304, Santa Barbara, CA, 1995.

[OPTZ96] A.J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice & Experience*, 26(2), Fevereiro de 1996.

[ORZ93] A.J. Offutt, G. Rothermel, and C. Zapt. An experimental evaluation of selective mutation. In *15th Int'l Conf. on Software Engineering*, pages 100–107, Baltimore - MD, Maio de 1993.

[OW91] T.J. Ostrand and E.J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Fourth Symp. Soft. Testing, Analysis, and Verification*, pages 74–86, Outubro de 1991.

[PBC93] A.S. Parrish, R.B. Borie, and D.W. Cordes. Automated flow graph - based testing of object-oriented software modules. *Journal of Systems Software*, 23:95–109, 1993.

[PLR94] H.D. Pande, W.A. Landi, and B.G. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Trans. Soft. Eng.*, 20(5), Maio de 1994.

[Pre92] R.S. Pressman. *Software Engineering: A Practitioner's Approach.* McGRAW-HILL, 1992.

[RH96] G. Rothermel and M.J. Harrold. Analysing regression test selection techniques. *IEEE Trans. Soft. Eng.*, 22(8):529–551, Agosto de 1996.

[Rin96] D. Rine. Structural defects in object-oriented programming. *ACM SIGSOFT – Software Engineering Notes*, 21(2):86–88, Março de 1996.

[RW82] S. Rapps and E.J. Weyuker. Data flow analysis techniques for test data selection. In *International Conference on Software Engineering*, pages 272–278, Tokio - Japan, Setembro de 1982.

[RW85] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Soft. Eng.*, SE-11(4), Abril de 1985.

[SC96] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Trans. Soft. Eng.*, 22(11), Novembro de 1996.

[Sil94] J.B. Silva. Protest+ ambiente de validação automática de qualidade de software através de técnicas de teste e de métricas de complexidade. Dissertação de Mestrado, CPGCC - UFRGS, Porto Alegre – RS, Brasil 1994.

[SMV97] S.R.S Souza, J.C. Maldonado, and S.R. Vergilio. Análise de mutantes e potenciais usos: Uma avaliação empírica. In *VIII International Conference on Software Technology*, Curitiba - Brasil, Junho de 1997.

[Sou96] S.R.S. Souza. Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de programas. Dissertação de Mestrado, SCE-ICMSC-USP, São Carlos – SP, Brasil, Junho de 1996.

[SR92] M.D. Smith and D.J. Robson. A framework for testing object-oriented programs. *Journal of Object Oriented Programming*, pages 45–53, Junho de 1992.

[TD97] K-C Tai and F.J. Daniels. Test order for inter-class integration of object-oriented software. In *21st Annual Int. Computer Software and Applications Conference (COMPSAC'97)*, pages 602–607, Washington, DC, Agosto de 1997.

[TLK92] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Soft. Eng.*, 18(3):206–215, Março de 1992.

[UY88] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28:157–163, 1988.

[VMJ95] S.R. Vergilio, J.C. Maldonado, and M. Jino. Um experimento de aplicação de critérios baseados em fluxo de dados no teste de programas c. In *XXII Seminário Integrado de Software e Hardware*, pages 941–952, 1995.

[VMJ97a] P. R. S. Vilela, J. C. Maldonado, and M. Jino. Program graph visualization. *Software-Practice and Experience*, 27(11), Novembro 1997.

[VMJ97b] P.R.S. Vilela, J.C. Maldonado, and M. Jino. Data flow based testing of programs with pointers: A strategy based on potential uses. In *10th International Software Quality Week 1997 – QW97*, San Francisco – CA, Maio de 1997.

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

[Wey88] E.J. Weyuker. An empirical study of the complexity of data flow testing. In *Second Workshop on Software, Verification and Analysis*, Banff, Canadá, Julho de 1988.

[Wey90] E.J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Trans. Soft. Eng.*, SE-16(2):121–128, Fevereiro de 1990.

[WHH80] M.R. Woodward, D. Heddley, and M.A. Hennel. Experience with path analysis and testing of programs. *IEEE Trans. on Software Engineering*, SE-6:278–286, Maio de 1980.

[WJ91] E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Soft. Eng.*, 17(7):703–711, Julho de 1991.

[WM95a] W.E. Wong and A.P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, Março de 1995.

[WM95b] W.E. Wong and A.P. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, Dezembro de 1995.

[WMM94] W.E. Wong, A.P. Mathur, and J.C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In *First IFIP/SQI - International Conference on Software Quality and Productivity (ICSQP'94): Theory, Practice, Education and Training*, Hong Kong, Dezembro de 1994.

[WO80] E.J. Weyuker and T.J Ostrand. Theory of program testing and the application of revealing subdomains. *IEEE Trans. Soft. Eng.*, SE-6, 1980.

[Won93] W.E. Wong. *On Mutation and Data Flow.* PhD thesis, Dept. Computer Science – Purdue University, W. Lafayette – IN, USA, Dezembro de 1993.

[WS85] L.J. White and P.N. Sahay. A computer system for generating test data using the domain strategy. In *Softfair II*, pages 38–45, San Francisco, CA, Dezembro de 1985.

Apêndice A

Teste de Unidade

Este apêndice apresenta os Modelos de Implementação dos Critérios Potenciais Usos para o Teste de Unidade e as principais informações geradas pela ferramenta POKE-TOOL durante o teste de unidade. A informação apresentada neste apêndice foi extraída de [Mal91] e [Cha91] ou geradas automáticamente pela ferramenta POKE-TOOL, não constituindo por sí só uma contribuição desta tese.

A.1 Modelos de Implementação para o Teste de Unidade

A ferramenta POKE-TOOL [MCJ89, Cha91] implementa os Critérios Potenciais Usos, originalmente definidos por Maldonado [Mal91] e apresentados no Capítulo 3, para o Teste de Unidade de programas. A ferramenta tem uma arquitetura multi-linguagem baseada na utilização de uma *linguagem intermediária* – a *LI*; está atualmente configurada para as linguagens C, Pascal, COBOL, FORTRAN e Clipper.

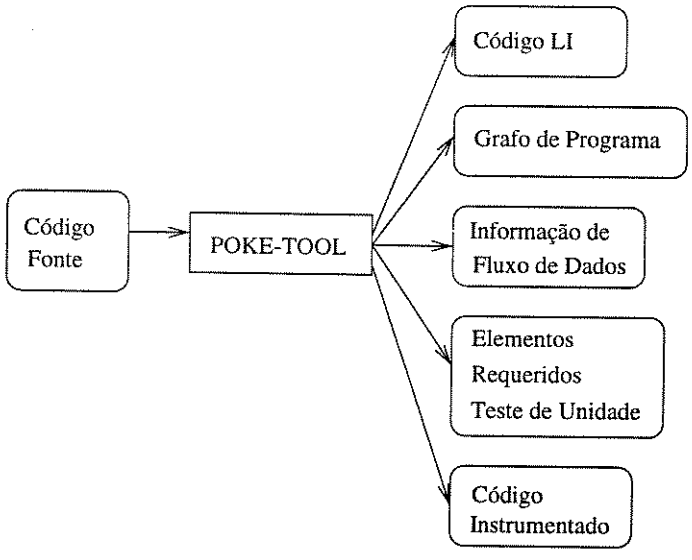

Figura A.1: Informação Gerada pela POKE-TOOL

A ferramenta recebe o código fonte do programa como entrada e gera a versão na linguagem intermediária; a partir daí ela determina o grafo de programa. Esse grafo é estendido incorporando-se informações a respeito do fluxo de dados do programa, gerando o chamado *grafo-def*; alguns recursos de otimização são aplicados para reduzir o número

de requisitos de teste identificados; um exemplo é o algoritmo REHFLUXDA para redução de herdeiros de fluxo de dados [Mal91]. O algoritmo REHFLUXDA fornece a lista de *arcos primitivos* utilizados para construir *descritores* (expressões regulares) dos caminhos ou associações de fluxo de dados requeridas pelos Critérios Potenciais Usos. Uma *versão instrumentada* do programa em teste também é gerada. Essa versão contém comandos adicionais, normalmente de escrita em arquivo, para rastrear o programa durante a sua execução; neste caso, escrevendo em um arquivo específico os nós percorridos durante uma execução – essa informação é chamada de "trace" do programa. Um resumo das informações geradas pela POKE-TOOL é mostrado na Figura A.1. Um exemplo é apresentado no Apêndice A.2.

A instrumentação informa os caminhos efetivamente percorridos durante a execução do programa com os casos de teste fornecidos pelo testador. Já os descritores servem para verificar (avaliar) se o critério selecionado foi satisfeito pelos casos de teste fornecidos; essa verificação é feita através de um aceitador de expressões regulares (máquina de estados).

Nesta seção são apresentados alguns aspectos teóricos usados no projeto e implementação da ferramenta POKE-TOOL; considera-se a utilização de critérios de teste estruturais como critérios de adequação de dados de teste, ou seja, usá-los para se saber quanto dos requisitos estabelecidos pelo critério para um determinado programa foi satisfeito pelos dados de teste; neste caso, funções como a instrumentação da unidade em teste, a monitoração dos caminhos executados pelo conjunto de casos de teste fornecidos pelo testador e a determinação dos elementos requeridos executados e não executados, devem ser incorporadas à ferramenta.

A.1.1 Modelo de Fluxo de Controle

No modelo de fluxo de controle adotado, um programa P é representado por um *grafo de programa* (como definido na Terminologia da Seção 2.1), onde os blocos disjuntos de comandos são associados aos *nós* e os possíveis fluxos de controle são associados aos *arcos*. Os blocos disjuntos de comandos têm a seguinte propriedade: uma vez executado o primeiro comando do bloco, os demais comandos são também executados na ordem dada.

A Figura A.2 representa as construções básicas do grafo de programa adotada para os comandos da linguagem LI, a menos dos comandos de desvios incondicionais [Mal91]. O grafo de programa de um módulo é obtido pela concatenação dessas construções básicas.

A.1.2 Modelo de Instrumentação

A instrumentação do código fonte tem como objetivo inserir modificações no programa em teste que permitam o seu rastreamento durante a execução. Desta forma, pode-se depois da execução dos casos de teste proceder à análise de adequação desse conjunto de casos de teste. No caso da POKE-TOOL a unidade instrumentada é armazenada no arquivo *testeprog.c*.

A instrumentação consiste em inserir *pontas de provas* nos blocos de comandos correspondentes a cada nó do grafo de programa da unidade em teste. Uma ponta de prova, nesse caso, é um comando de escrita em arquivo que escreve o número correspondente ao bloco executado, isso permite a identificação do caminho executado pelo caso de teste fornecido.

A descrição detalhada do modelo de instrumentação utilizado na POKE-TOOL pode ser encontrado em [Mal91].

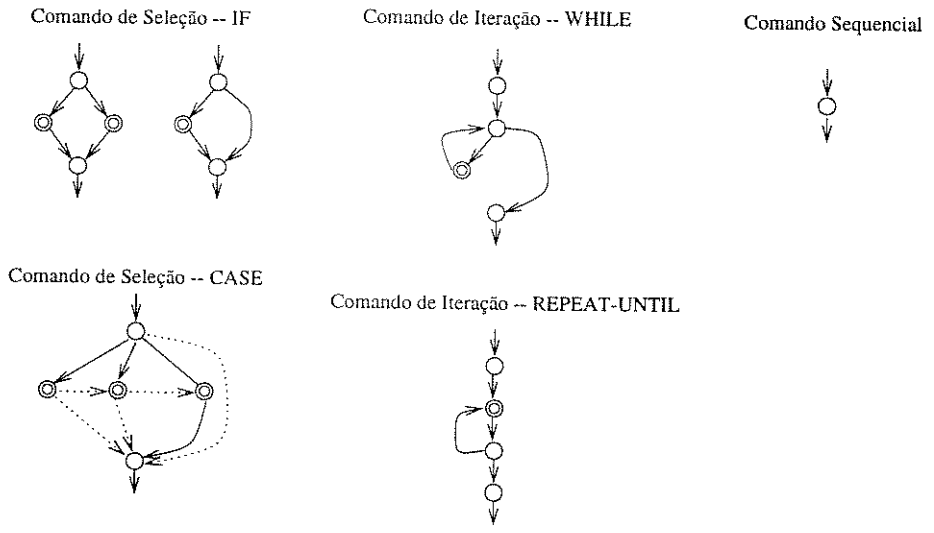

Figura A.2: Modelo de Fluxo de Controle.

A.1.3 Modelo de Fluxo de Dados

No contexto de teste de software, a análise de fluxo de dados é utilizada para se adicionar à informação de fluxo de controle a informação sobre os tipos de ocorrências de variáveis. Essa informação é posteriormente utilizada para se determinar os elementos requeridos de cada um dos critérios baseados em análise de fluxo de dados. Basicamente as ocorrências de variáveis em um programa podem ser de três tipos: uma *definição*, um *uso* ou uma *indefinição*.

O *uso* de uma variável ocorre quando o seu valor é lido da posição de memória que a armazena; esse tipo de uso não precisa ser identificado na implementação dos Critérios Potenciais Usos, já que esses critérios não exigem a ocorrência explícita de um uso de variável para estabelecer um requisito de teste.

Para os Critérios Potenciais Usos é suficiente associar a cada nó i do grafo de programa o conjunto de variáveis definidas no bloco de comandos correspondente; ao conjunto formado pela informação de fluxo de controle estendido com informação de fluxo de dados dá-se o nome de *grafo def*.

Uma *definição* de variável ocorre quando um valor é armazenado em uma posição de memória; em geral, a ocorrência de uma variável é uma definição se ela: i) aparece no lado esquerdo de um comando de atribuição; ii) está em um comando de entrada; ou iii) é passada como parâmetro de saída em comandos de chamada a procedimentos.

Uma distinção importante acontece no tratamento de variáveis passadas como parâmetro. A passagem de valores entre procedimentos através de passagem de parâmetros pode ser feita por: *valor*, *referência* ou *nome* [GJ87]. Se a variável for passada por referência ou nome, considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são distinguidas das demais e chamadas de *definidas por referência*; esta distinção é utilizada na geração dos *grafos(i)* (ver próxima subseção), ou seja, na determinação dos caminhos livres de definição para as variáveis definidas no nó i. Essa abordagem é também utilizada para o tratamento de variáveis do tipo vetor, e no Capítulo 3 analisa-se sua aplicação para o tratamento de variáveis do tipo ponteiro. De forma resumida, a definição por referência de uma variável no nó i é considerada para a determinação das associações de fluxo de dados a partir do nó i até todos

os pontos alcançáveis por caminhos livre de definição, mas essa definição por referência não é considerada para restringir a busca por possíveis usos de uma definição anterior, ou seja, ela não é considerada uma redefinição de variável. Essa abordagem é conservadora já que não se pode afirmar com certeza, através de uma análise estática, se uma dada variável foi ou não redefinida em uma chamada de procedimento [Mal91].

Essa distinção no tratamento de variáveis definidas por referência influencia a abordagem de teste de programas com ponteiros apresentada do Capítulo 4, já que a mesma abordagem conservadora é utilizada no tratamento de variáveis redefinidas através de dereferência por ponteiro. Esse tratamento não tem influência direta na abordagem para o Teste de Integração, já que as associações que surgem dada a característica conservadora da análise não correspondem diretamente à integração entre dois módulos, mas pode haver uma influência indireta quando uma associação que morreria numa chamada à procedimento atinge outra chamada a procedimento.

Uma *indefinição* de variável ocorre quando 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 de um módulo; neste sentido o nó de saída tem uma indefinição de todas as variáveis locais.

A.1.4 Modelo de Descrição dos Elementos Requeridos

O *Modelo de Descrição dos Elementos Requeridos* pelos Critérios Potenciais Usos foi definido por Maldonado [Mal91] com o objetivo de dar uniformidade à implementação de ferramentas de teste estrutural, em especial aos Critérios Potenciais Usos. Este modelo é fundamentado no conceito de arco primitivo [Chu87] e no conceito de Grafo(i) [MCJ88, Mal91].

O conceito de arco primitivo se baseia no fato de existirem arcos, em um grafo de fluxo de controle, que são sempre executados quando um outro arco é executado; esses arcos são ditos não essenciais para a análise de cobertura. O algoritmo que viabiliza a utilização do conceito de arco primitivo no contexto de teste baseado em análise de fluxo de dados, incluindo as modificações ao algoritmo original proposto por Chusho foi apresentado por Maldonado et al. [MCJ88].

Com o objetivo de dar uniformidade à implementação de ferramentas de suporte a testes estruturais, em particular aos Critérios Potenciais Usos, foi sugerido o conceito de *grafo(i)* [MCJ88], obtido a partir do grafo def. O grafo(i) fornece "todos" os caminhos livres de definição c.r.a qualquer variável definida em i. A proposta é construir um grafo(i) para cada nó i onde houver definição de variável, obtendo-se por construção uma minimização de caminhos e associações requeridos.

A partir dos grafos(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. Maldonado e Chaim [Mal91, Cha91] apresentam uma descrição detalhada dos descritores para os Critérios Potenciais Usos. Segue, como exemplo, a caracterização dos descritores para o Critérios Todos-Potenciais-DU-Caminhos.

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 grafos(i). Cada um desses caminhos é identificado por uma sequência de arcos primitivos.

Seja um caminho π do grafo(i) constituído pelos arcos primitivos: $p_1 = (k_1, l_1), p_2 = (k_2, l_2), \cdots, p_n = (k_n, l_n)$.

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

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

Se $k_1 = i$, o descritor desse caminho é:

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

No descritor, N é o conjunto de todos os nós do grafo de programa e N_{lf} é um conjunto de nós com características dinâmicas, isto é, cada vez que um nó deste conjunto é percorrido ele é retirado do conjunto; essa estrutura é utilizada para garantir que um caminho livre de laço seja exercitado ("loop free"). O símbolo "*" indica que qualquer elemento dos conjuntos N ou N_{lf} poderá ocorrer a menos do elemento correspondente ao número imediatamente seguinte a $N*$ ou $N_{lf}*$, quando ocorre uma transição de estado. N_{lf} é usado nos descritores para o Critérios Todos-Potenciais-DU-Caminhos.

Esses descritores – expressões regulares – constituem a base para a construção dos aceitadores utilizados na avaliação da adequação de um dado conjunto de casos de teste.

A.2 Informação Gerada pela POKE-TOOL

A motivação para se utilizar uma linguagem intermediária no contexto de ferramentas de teste de programas é que a informação mais importante que se quer abstrair de um programa em teste é o seu fluxo de controle ou de execução. Por isso pode-se identificar, basicamente, dois tipos de comandos na *Linguagem Intermediária* (LI) definida para a POKE-TOOL: os comandos *sequenciais* e os comandos de *controle de fluxo*. Os comandos sequenciais da LI indicam os comandos das linguagens procedimentais que representam uma declaração de variável ou uma computação que, portanto, não alteram o fluxo de execução do programa. Os comandos de controle de fluxo da LI são equivalentes aos comandos das linguagens procedimentais que causam *seleção*, *seleção múltipla*, *iteração* e *transferência incondicional*. A partir da identificação dos comandos sequenciais e dos comandos de controle de fluxo de um programa pode-se gerar a *versão instrumentada do programa* e construir seu *grafo de fluxo de controle* (ou *grafo de programa*).

Observa-se no exemplo a seguir os arquivos gerados pela ferramenta POKE-TOOL a partir do programa fonte original até obter-se o grafo de programa e a versão instrumentada. Considere o programa *compress.c*, transcrito para C do original em PASCAL publicado por B. W. Kernighan e P. J. Plauger [KP81].

```
void compress(){
  int n,lastc,c;
  n = 1;
  lastc = getchar();
  while (lastc != ENDFILE){
   if ((c=getchar()) == ENDFILE){
     if ((n>1)||(lastc==WARNING))
       putrep(n,lastc);
     else
```

```
        putchar(lastc);
    }
    else{
     if (c==lastc)
       n++;
     else if ((n>1)||(lastc==WARNING)){
            putrep(n,lastc);
            n = 1;
          }
          else
            putchar(lastc);
    }
    lastc = c;
  }
}
```

Esse programa é traduzido para a linguagem intermediária considerando-se apenas declarações de variáveis: "$DCL", comandos sequenciais: "$S", comando de controle de fluxo: "$REPEAT", "$UNTIL", "$WHILE" e "$IF", "$ELSE", com as respectivas condições: "$C" e "$NC". Os números que aparecem logo depois do átomo em cada linha da LI representam respectivamente: o número do nó, o início do comando, o comprimento e a linha em que o comando aparece; na verdade essa informação promove a ligação da LI com o programa fonte original. Quando um programa tem mais de um módulo a ferramenta POKE-TOOL pode gerar a LI de todos os módulos de uma só vez e deixá-los em um único arquivo, essa é uma característica útil para se implementar o teste de integração.

```
@compress          1     0     0     0
$DCL               1    20    15     4
{                  1   288     1    11
$DCL               1   292    14    12
$S01               1   310     6    14
$S02               1   319    18    15
$WHILE             2   340     5    16
$C(01)01           2   346    18    16
{                  3   367     1    17
$IF                3   372     2    18
$C(01)02           3   375    26    18
{                  4   405     1    19
$IF                4   412     2    20
$C(02)03           4   415    25    20
{                  5     0     0     0
$S03               5   448    16    21
}                  5     0     0     0
$ELSE              6   470     4    22
{                  6     0     0     0
$S04               6   482    15    24
}                  6     0     0     0
}                  7   501     1    25
$ELSE              8   506     4    26
{                  8   514     1    28
$IF                8   520     2    29
$C(01)04           8   523    10    29
{                  9     0     0     0
$S05               9   540     4    30
}                  9     0     0     0
```

```
$ELSE                10    549     4    31
{                    10      0     0     0
$IF                  10    554     2    31
$C(02)05             10    557    25    31
{                    11    592     1    32
$S06                 11    605    16    33
$S07                 11    633     6    34
}                    11    649     1    35
$ELSE                12    660     4    36
{                    12      0     0     0
$S08                 12    676    15    38
}                    12      0     0     0
}                    13      0     0     0
}                    14    695     1    39
$S09                 15    700    10    40
}                    15    713     1    41
}                    16    715     1    42
```

Um efeito colateral do uso de uma linguagem intermediária em uma ferramenta de teste é a possibilidade de transformá-la em uma ferramenta multi-linguagem. Esse é o caso da POKE-TOOL; para adaptar a ferramenta para uma nova linguagem basta construir o tradutor da linguagem fonte para a linguagem intermediária usada na ferramenta; o núcleo da ferramenta não precisa ser alterado.

A seguir o programa fonte original é acrescido de comandos de escrita em arquivo que vão gerar o "trace" do programa. Essa é a versão instrumentada do programa. Quando essa nova versão do programa for compilada e executada para os casos de teste gerados pelo testador, os comando de escrita em arquivo nela inseridos vão fornecer os caminhos percorridos durante a execução de cada um dos casos de teste.

```
void compress(){
                    FILE * path = fopen("compress/path.tes","a");
                    static int printed_nodes = 0;
/*  1 */            int n,lastc,c;
                    ponta_de_prova(1);
/*  1 */            n = 1;
/*  1 */            lastc = getchar();
/*  2 */            while(lastc != ENDFILE){
                       ponta_de_prova(2);
                       ponta_de_prova(3);
/*  3 */               if((c=getchar()) == ENDFILE){
                          ponta_de_prova(4);
/*  4 */                  if((n>1)||(lastc==WARNING)){
                             ponta_de_prova(5);
/*  5 */                     putrep(n,lastc);
/*  5 */                  }
/*  6 */                  else{
                             ponta_de_prova(6);
/*  6 */                     putchar(lastc);
/*  6 */                  }
                          ponta_de_prova(7);
/*  7 */               }
/*  8 */               else{
                          ponta_de_prova(8);
/*  8 */                  if(c==lastc){
                             ponta_de_prova(9);
```

```
/*  9 */                    n++;
/*  9 */                 }
/* 10 */                 else{
                            ponta_de_prova(10);
/* 10 */                    if((n>1)||(lastc==WARNING)){
                               ponta_de_prova(11);
/* 11 */                       putrep(n,lastc);
/* 11 */                       n = 1;
/* 11 */                    }
/* 12 */                    else{
                               ponta_de_prova(12);
/* 12 */                       putchar(lastc);
/* 12 */                    }
                            ponta_de_prova(13);
/* 13 */                 }
                         ponta_de_prova(14);
/* 14 */              }
                      ponta_de_prova(15);
/* 15 */              lastc = c;
/* 15 */           }
                   ponta_de_prova(2);
                   ponta_de_prova(16);
                   fclose(path);
/* 16 */        }
```

A partir da LI também gera-se o grafo de programa inicialmente em forma textual como é mostrado a seguir, ou então graficamente, através da ferramenta ViewGraph [VMJ97a], como mostra a Figura A.3.

```
16

1
   2 0
2
   3 16 0
3
   4 8 0
4
   5 6 0
5
   7 0
6
   7 0
7
   15 0
8
   9 10 0
9
   14 0
10
   11 12 0
11
   13 0
12
   13 0
13
```

```
   14 0
14
   15 0
15
   2 0
16
   0
```

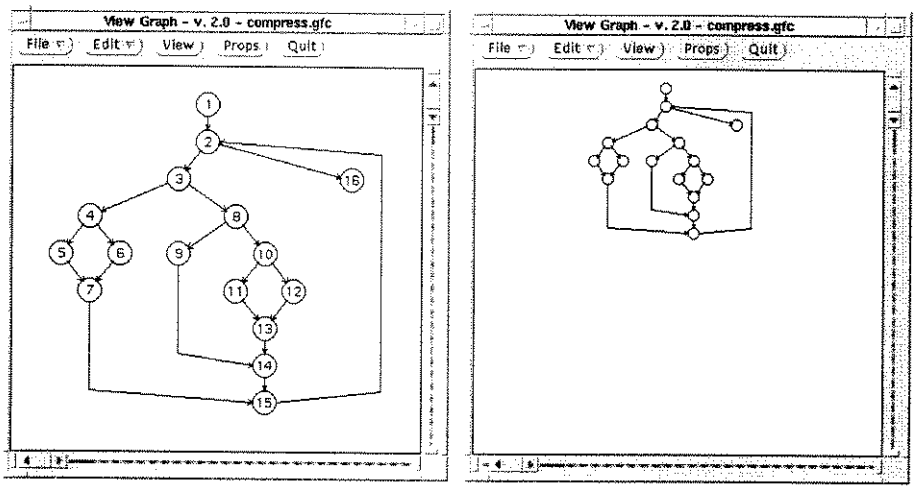

Figura A.3: Grafo do Programa Compress()

Associando-se a cada nó do grafo de programa as variáveis que foram definidas naquele nó obtém-se a informação de fluxo de dados pertinente aos Critérios Potenciais Usos. O grafo acrescido de informações de fluxo de dados é chamado *grafo-def*. Esse grafo fornece as informações essenciais para a análise de fluxo de dados segundo os Critérios Potenciais Usos; ou seja, os nós onde houve definição de variável, quais variáveis foram definidas e se essa definição foi uma definição por referência feita numa chamada de procedimento.

```
# Tabela de Variaveis Definidas do Modulo compress

# Globais

# Locais
0 2 c
0 1 lastc
0 0 n
00

# Grafo Def Sintetico do Modulo compress

00  1
Defs:   0   1 0
Refs: 0
00  2
Defs: 0
Refs: 0
00  3
Defs:   2 0
Refs: 0
```

```
@@  4
Defs: @
Refs: @
@@  5
Defs: @
Refs: @
@@  6
Defs: @
Refs: @
@@  7
Defs: @
Refs: @
@@  8
Defs: @
Refs: @
@@  9
Defs:   0 @
Refs: @
@@ 10
Defs: @
Refs: @
@@ 11
Defs:   0 @
Refs: @
@@ 12
Defs: @
Refs: @
@@ 13
Defs: @
Refs: @
@@ 14
Defs: @
Refs: @
@@ 15
Defs:   1 @
Refs: @
@@ 16
Defs: @
Refs: @
```

Com a informação a respeito do fluxo de dados do programa monta-se a lista de elementos requeridos para cada um dos critérios implementados na ferramenta POKE-TOOL. A lista a seguir, por exemplo, corresponde às associações requeridas pelo *Critério Todos-Potenciais-Usos* ou pelo *Critério Todos-Potenciais-Usos/DU.*

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

Associacoes requeridas pelo Grafo( 1)
1) <1,(2,16),{ n, lastc }>
2) <1,(10,12),{ n, lastc }>
3) <1,(10,11),{ n, lastc }>
4) <1,(14,15),{ lastc }>
5) <1,(8,9),{ n, lastc }>
6) <1,(4,6),{ n, lastc }>
7) <1,(15,2),{ n }>
8) <1,(7,15),{ n, lastc }>
```

```
9) <1,(4,5),{ n, lastc }>

Associacoes requeridas pelo Grafo( 3)
10) <3,(10,12),{ c }>
11) <3,(10,11),{ c }>
12) <3,(8,9),{ c }>
13) <3,(4,6),{ c }>
14) <3,(2,16),{ c }>
15) <3,(2,3),{ c }>
16) <3,(4,5),{ c }>

Associacoes requeridas pelo Grafo( 9)
17) <9,(2,16),{ n }>
18) <9,(13,14),{ n }>
19) <9,(10,12),{ n }>
20) <9,(10,11),{ n }>
21) <9,(8,9),{ n }>
22) <9,(4,6),{ n }>
23) <9,(7,15),{ n }>
24) <9,(4,5),{ n }>

Associacoes requeridas pelo Grafo(11)
25) <11,(2,16),{ n }>
26) <11,(12,13),{ n }>
27) <11,(10,12),{ n }>
28) <11,(10,11),{ n }>
29) <11,(8,9),{ n }>
30) <11,(4,6),{ n }>
31) <11,(7,15),{ n }>
32) <11,(4,5),{ n }>


Associacoes requeridas pelo Grafo(15)
33) <15,(2,16),{ lastc }>
34) <15,(10,12),{ lastc }>
35) <15,(10,11),{ lastc }>
36) <15,(14,15),{ lastc }>
37) <15,(8,9),{ lastc }>
38) <15,(4,6),{ lastc }>
39) <15,(7,15),{ lastc }>
40) <15,(4,5),{ lastc }>
```

Cada associação tem um descritor. Os Descritores são usados por aceitadores de caminhos para decidir quais elementos requeridos foram efetivamente cobertos pelos casos de teste. Os descritores dos caminhos e associações requeridos pelos Critérios Potenciais Usos são autômatos finitos que devem ser satisfeitos para que o caminho ou associação possa ser considerada exercitada. Para tanto deve ser exercitado pelos casos de teste um caminho que realize as transições de estados dentro do autômato para levá-lo do *estado inicial* para o *estado final.*

Para descrever, por exemplo, uma associação $< 1, (2, 16), \{n, lastc\} >$ requerida pelo Critério Todos-Potenciais-Usos é utilizado o seguinte descritor:

$$N * 1\ N_{nv} * 2\ [N_{nv} * 2] * 16.$$

N_{nv} é o conjunto de nós do $grafo(i)$ [1], com exceção daqueles nós onde ocorre definição de v. N_{nv} é usado nos descritores do Critérios Todos-Potenciais-Usos.

Os descritores do Critério Todos-Potenciais-Usos/DU são mais simples. Considere a associação $< 2, (9, 14), newcol >$ o descritor é:

$$N * \ 2 \ N_{nvlf} * \ 9 \ 14.$$

O conjunto N_{nvlf} é, inicialmente, igual a N_{nv} porém é alterado dinamicamente como o conjunto N_{lf}.

Encontra-se a seguir a lista de descritores do Critério Todos-Potenciais-Usos para o programa compress (N_t representa os estados terminais).

```
DESCRITORES PARA O CRITERIO TODOS POT-USOS

N = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Descritores para o Grafo( 1)
Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Nt = 2 15 16

1) N* 1 Nnv* 2 [ Nnv* 2 ]* 16
Nnv = 2 3 4 5 6 7 8 10 12 13 14 16
2) N* 1 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 5 6 7 8 10 12 13 14 16
3) N* 1 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 5 6 7 8 10 12 13 14 16
4) N* 1 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
5) N* 1 Nnv* 8 [ Nnv* 8 ]* 9
Nnv = 2 3 4 5 6 7 8 10 12 13 14 16
6) N* 1 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 5 6 7 8 10 12 13 14 16
7) N* 1 Nnv* 15 [ Nnv* 15 ]* 2
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
8) N* 1 Nnv* 7 [ Nnv* 7 ]* 15
Nnv = 2 3 4 5 6 7 8 10 12 13 14 16
9) N* 1 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 5 6 7 8 10 12 13 14 16

Descritores para o Grafo( 3)
Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Nt = 3 16

10) N* 3 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15 16
11) N* 3 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15 16
12) N* 3 Nnv* 8 [ Nnv* 8 ]* 9
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15 16
13) N* 3 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15 16
```

[1] $Grafo(i)$ [MCJ88, Mal91] é o grafo que contém a partir do nó i todos os sub-caminhos alcançáveis sem redefinição das variáveis definidas em i.

```
14) N* 3 Nnv* 2 [ Nnv* 2 ]* 16
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15 16
15) N* 3 Nnv* 2 [ Nnv* 2 ]* 3
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15 16
16) N* 3 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15 16

Descritores para o Grafo( 9)
Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Nt = 9 11 14 15 16

17) N* 9 Nnv* 2 [ Nnv* 2 ]* 16
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
18) N* 9 Nnv* 13 [ Nnv* 13 ]* 14
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
19) N* 9 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
20) N* 9 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
21) N* 9 Nnv* 8 [ Nnv* 8 ]* 9
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
22) N* 9 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
23) N* 9 Nnv* 7 [ Nnv* 7 ]* 15
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
24) N* 9 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16

Descritores para o Grafo(11)
Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Nt = 9 11 13 15 16

25) N* 11 Nnv* 2 [ Nnv* 2 ]* 16
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
26) N* 11 Nnv* 12 [ Nnv* 12 ]* 13
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
27) N* 11 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
28) N* 11 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
29) N* 11 Nnv* 8 [ Nnv* 8 ]* 9
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
30) N* 11 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
31) N* 11 Nnv* 7 [ Nnv* 7 ]* 15
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16
32) N* 11 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 5 6 7 8 10 12 13 14 15 16

Descritores para o Grafo(15)
Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Nt = 15 16

33) N* 15 Nnv* 2 [ Nnv* 2 ]* 16
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
34) N* 15 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
```

```
35) N* 15 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
36) N* 15 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
37) N* 15 Nnv* 8 [ Nnv* 8 ]* 9
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
38) N* 15 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
39) N* 15 Nnv* 7 [ Nnv* 7 ]* 15
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16
40) N* 15 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 16

Numero Total de Descritores = 40
```