POKE-TOOL – Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados

Marcos Lordello Chaim

Orientador: Prof. Dr. Mario Jino

Dissertação apresentada à Faculdade de Engenharia Elétrica da UNICAMP, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica.

Abril de 1991

À minha avó Cheterrim Abirad Chaim Cassab (Cheter).

Agradecimentos

Inicialmente, gostaria de agradecer ao meu orientador, Prof. Dr. Mario Jino, pela oportunidade de trabalho, pela amizade e orientação desde 1985, e também por despertar em mim o gosto pela Engenharia de Software.

Ao José Carlos Maldonado, pelas idéias e motivação, sem as quais a POKE-TOOL não seria "a" ferramenta de suporte à aplicação dos Critérios Potenciais Usos. E, principalmente, pela amizade e paciência de enxergar as deficiências como virtudes.

A todo o Grupo de Teste de Software do Departamento de Engenharia da Computação e Automação Industrial (DCA): Mauro Carnassale, Sérgio Augusto Mota Caracas, Sílvia Regina Vergílio, Rubens Pontes Fonseca e Plínio de Sá Leitão Júnior. Em especial, gostaria de agradecer à Sílvia pela compreensão com que enfrentou, junto com o Maldonado, as primeiras versões da ferramenta.

A todos meus amigos e colegas de pós-graduação do DCA e da Faculdade de Engenharia Elétrica da Unicamp. Ao Heraldo França Madeira pelas inúmeras "dicas" fornecidas durante o desenvolvimento deste trabalho. Ao Augusto Cézar Freitas de Morais que não poupou seus próprios meios (e da "Big Blue"!) para fornecer o suporte logístico necessário.

Aos meus amigos do Instituto de Biociências, Letras e Ciências Exatas da UNESP, S. J. do Rio Preto. Em especial, ao Prof. Dr. Odelar Leite Linhares e aos meus amigos Antônio Fernando Fortes Castelo Branco e Farid Nourani.

À Luísa A. Spadacini Laera do Instituto de Ciências Matemáticas de São Carlos que gentilmente confeccionou as figuras em LaTeX.

À Cris, cuja presença e carinho tornou tudo mais fácil desde o começo.

Aos meus pais Aziz e Elza; aos meus irmãos Baca, Tó (Mariângela e quem vai chegar...) e Cassinha. Enfim, à grande família Cassab Chaim, que sempre acreditou em tudo.

À CAPES, CNPq e UNESP pelo suporte financeiro.

Sumário

Os principais aspectos da especificação e implementação de uma ferramenta multilinguagem para suporte ao teste estrutural de programas baseado em fluxo de dados são apresentados. Na versão atual a ferramenta, denominada POKE-TOOL, suporta o teste de programas escritos na linguagem C e automatiza a aplicação dos critérios Potenciais Usos (PU) [MAL88a, MAL88b].

Os pontos mais relevantes e os principais algoritmos da implementação são apresentados em detalhe. São também descritos os passos do procedimento a ser realizado por um usuário configurador para gerar configurações desta ferramenta para outras linguagens procedurais. Os aspectos funcionais e de controle de atividades da POKE-TOOL são ilustrados através de uma sessão de trabalho completa, que mostra a aplicação da ferramenta em um programa; o programa exemplo foi extraído de um conjunto de programas utilizado para conduzir um "benchmark" dos critérios Potenciais Usos.

Abstract

The main aspects of the specification and implementation of a multilanguage tool for structural data flow testing of programs are presented. In the present version, the tool, named POKE-TOOL, supports the test of programs written in C; it automates the application of the Potential Uses Criteria [MAL88a, MAL88b].

The most relevant points and main algorithms of the implementation are presented in detail. We also describe the steps of the procedure to be carried out by a user-configurer to generate configurations of the tool for other procedural languages. Functional and activities control aspects of POKE-TOOL are ilustrated through a complete work session, showing the application of the tool on a program; the example program was extracted from a set of programs used to conduct a benchmark of the Potential Uses criteria.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Trabalhos Relacionados

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

[CHA91a] M. L. Chaim, J. C. Maldonado e Mario Jino, *Manual de Configuração da POKE-TOOL*, *Relatório Técnico* DCA/RT/008/91 - DCA/FEE/UNICAMP - Fev. 1991.

[CHA91b] M. L. Chaim, J. C. Maldonado e Mario Jino, *Manual do Usuário da POKE-TOOL*, *Relatório Técnico* DCA/RT/009/91 - DCA/FEE/UNICAMP - Fev. 1991.

[CHA91c] M. L. Chaim, J. C. Maldonado e Mario Jino, *Projeto de uma Ferramenta de Teste de Programas*, *Relatório Técnico* DCA/RT/010/91 - DCA/-FEE/UNICAMP - Fev. 1991.

[MAL88a] J. C. Maldonado, M. L. Chaim, M. Jino, " Seleção de Casos de Testes Baseada em Fluxo de Dados através dos Critérios Potenciais Usos," *in Proc. II Simp. Bras. Eng. de Software*, Canela, R.S., pp. 24-35, Out. 1988.

[MAL88b] J. C. Maldonado, M. L. Chaim, M. Jino, "Resultados do estudo de uma família de critérios de teste de programas baseado em fluxo de dados," *Relatório Técnico* DCA/RT/001/88 - DCA/FEE/UNICAMP - Dez. 1988.

[MAL89a] J. C. Maldonado, M. L. Chaim, M. Jino, "Arquitetura de Uma Ferramenta de Teste de Software de Apoio ao Critérios Potenciais Usos," *in Proc. XXII Congresso Nacional de Informática*, São Paulo, S.P., pp. 92-101, Set. 1989.

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

[**MAL91a**] J. C. Maldonado, M. L. Chaim, M. Jino, "Potential Uses Criteria Complexity Analysis," *Techinical Notes* - TN/DCA-001/91 - Campinas, SP, Brazil, 1991.

[**MAL91b**] J. C. Maldonado, M. L. Chaim, M. Jino, "Potential Uses Criteria: A Step towards Bridging the Gap in the Presence of Infeasible Paths," submetido para publicação, Março 1991.

[**MAL91c**] J. C. Maldonado, M. L. Chaim, M. Jino, "Using the Essential Branch Concept to Support Data-flow Based Testing Criteria Application," submetido para publicação, Março 1991.

Capítulo 1

Introdução

1.1 Necessidade de Ferramentas para Auxílio ao Teste de Software

Uma das metas da Engenharia de Software é produzir software de alta qualidade [PRE87]. Teste de software é uma das atividades de garantia de qualidade de software; de acordo com Chusho [CHU87] é a chave para aprimorar a produtividade e a confiabilidade do software. Teste de software envolve: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes [DEU82]. O projeto de casos de testes concentra-se em um conjunto de técnicas, critérios e métodos para elaborar os casos de teste. Estes métodos, critérios e técnicas fornecem ao projetista de software uma abordagem sistemática aos testes; além disto, constituem um mecanismo que pode auxiliar a garantir a completude dos testes e uma maior probabilidade de revelar defeitos no software. Em geral, deve-se projetar casos de testes que tenham a maior probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço.

Os métodos utilizados para testar o software são baseados em duas técnicas: funcional e estrutural. A técnica funcional procura selecionar casos de teste apoiada na especificação funcional do software enquanto a técnica estrutural apóia-se essencialmente no fluxo (estrutura) de controle [CHU87, MYE79]; a estrutura de controle ou grafo de controle é simplesmente uma notação para representar o fluxo de controle lógico de uma unidade de programa e consiste basicamente em um grafo dirigido [MCC76]. Independentemente das desvantagens dos critérios e métodos de teste estrutural [NTA88, RAP85], estes devem ser vistos como complementares aos funcionais, uma vez que cobrem essencialmente classes distintas de erros [MYE79, PRE87].

Com respeito ao teste estrutural vários critérios para a seleção de casos de teste foram estabelecidos. Os primeiros critérios a surgirem foram baseados unicamente

no fluxo de controle dos programas. Assim, têm-se critérios que requerem que todo o bloco de comandos sequenciais de um programa seja executado pelo menos uma vez, ou todos os comandos de transferência, ou mesmo todos os caminhos do programa. Note-se, entretanto, que este último critério pode levar a um número infinito de casos de teste.

Mais recentemente, vários outros critérios de testes estruturais baseados na análise de fluxo de dados [HEC77] de um programa foram desenvolvidos [HER76, LAS83, NTA84, RAP85, MAL88a, MAL88b, URA88]. A análise de fluxo de dados foi inicialmente utilizada para otimização de código por compiladores e, em geral, estabelece que a ocorrência de uma variável pode ser de dois tipos: definição e uso. No contexto de teste de software, métodos e critérios de projeto de casos de teste baseados em análise de fluxo de dados requerem que as interações que envolvem definições de variáveis de programa e subsequentes referências a essas definições sejam testadas. Portanto, esses critérios baseiam-se nas associações entre uma definição de uma variável e os seus possíveis subsequentes usos para a derivação de casos de teste.

Especificamente, Maldonado, Chaim e Jino [MAL88a, MAL88b], introduziram os Critérios Potenciais Usos baseados no conceito *potencial uso*; os Critérios Potenciais Usos são critérios de teste estrutural, baseados na análise de fluxo de dados e consistem, fundamentalmente, em variações da família de critérios apresentada por Rapps e Weyuker [RAP82, RAP85]; são denominados: critérios *todos-potenciais-du-caminhos*, *todos-potenciais-usos* e *todos-potenciais-usos/du*. Associações são requeridas independentemente da ocorrência explícita de uma referência a uma determinada definição; se um uso pode existir — *um potencial uso* — a potencial associação é requerida.

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

Essa situação ensejou o desenvolvimento de ferramentas automáticas para auxiliar na produção de testes efetivos e na análise dos resultados dos testes. As atividades típicas de uma ferramenta de teste são: fazer análise estática do código fonte, auxiliar na instrumentação do código, medir a cobertura fornecida por um conjunto de casos de teste e fornecer relatórios, por exemplo, indicando o número de vezes que cada comando ou sequência de comandos foi executado.

O uso de uma ferramenta de software para auxílio ao teste de programas pode ser vinculado a um critério de teste de duas maneiras. A ferramenta de software pode utilizar o critério de teste como um guia para a geração de casos de teste que

satisfaçam o critério; outra possibilidade é a utilização do critério para a análise de cobertura de um conjunto de casos de teste, isto é, verificar se os casos de teste aplicados preencheram os requisitos de teste do critério.

A aplicação dos critérios baseados em análise de fluxo de dados sem o apoio de uma ferramenta automatizada é limitada a programas muito simples [KOR85]. A importância de tais ferramentas tem sido discutida por muito outros autores [NTA88, FRA85, WEY84, PRI87, LAS83, WHI85] e algum esforço para implementar tal classe de ferramenta tem sido feito [HER76, FRA85, KOR85, MAL89, LUT90, PRI90].

Este trabalho descreve o projeto e implementação de uma ferramenta de suporte à aplicação dos critérios Potenciais Usos, denominada POKE-TOOL (POtential Uses CRIteria TOOL for Program Testing). A ferramenta POKE-TOOL está enquadrada na segunda abordagem das ferramentas de teste estrutural de programas. Ela fornecerá ao usuário os caminhos necessários para satisfazer os critérios Potenciais Usos e será capaz de verificar se o conjunto de casos de teste fornecido pelo usuário executou todos os caminhos requeridos. Diferentemente de suas antecessoras, a POKE-TOOL deverá ser uma ferramenta flexível, isto é, não atrelada a nenhuma linguagem de programação específica, permitindo que o usuário a configure para a linguagem de programação de seu interesse.

1.2 Trabalhos Relacionados

A partir da década de 70 começaram a surgir as primeiras ferramentas de apoio ao teste estrutural de software. Essas ferramentas suportavam, principalmente, a aplicação de critérios de teste baseados somente no grafo de fluxo de controle do programa a ser testado. No final da década de 70 e início dos anos 80, já surgiram as primeiras ferramentas de teste comerciais. Essas ferramentas suportavam basicamente a análise de cobertura dos critérios *todos-nós* (os comandos que fazem parte de cada nó do grafo de controle do programa em teste devem ser executados pelo menos uma vez) e *todos-ramos* (os comandos de transferência condicional devem ter todas suas condições executadas pelo menos uma vez), mais alguma forma de análise estática do programa fonte para detecção de anomalias de fluxo de dados e verificação dos tipos dos parâmetros utilizados nas chamadas de procedimentos. Essa última característica se deve ao fato dos compiladores da época não incorporarem esse tipo de análise do código fonte.

Na década de 80, com o surgimento dos critérios baseados em análise de fluxo de dados, começaram a surgir as ferramentas que suportam a aplicação desses critérios. Na sua maioria, essas ferramentas foram implementadas pelos criadores dos critérios de teste e visavam mostrar que a aplicação dessa nova classe de critérios era factível de ser utilizada na prática. Algumas experiências utilizando essas ferramentas em programas reais foram feitas [WEY88, WEY90, MAL91a] e

indicam que os critérios são passíveis de serem utilizados em ambiente industrial. Até o momento não está disponível comercialmente nenhuma ferramenta que utiliza critérios baseados em fluxo de dados; as ferramentas disponíveis são protótipos de pesquisa desenvolvidos em universidades.

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

A seguir descreveremos algumas das principais ferramentas de teste estrutural de software, principalmente aquelas que suportam a aplicação de critérios baseados em análise de fluxo de dados.

1.2.1 RXVP80

RXVP80 [DEU82] é um sistema que faz basicamente análise de cobertura do teste de ramos em programas escritos em FORTRAN. Além do suporte ao teste dinâmico, via instrumentação, essa ferramenta provê ainda: análise estática do código fonte com a geração do grafo de chamada dos módulos (unidades) constituintes do sistema em teste; geração do grafo de fluxo de controle dos módulos; verificação de anomalias no código fonte; verificação de tipos nas chamadas de procedimentos; e a geração de alguns relatórios que fornecem referências cruzadas entre código fonte, variáveis e módulos.

Ainda, com relação aos aspectos dinâmicos, RXVP80 faz uma instrumentação do código fonte que permite a obtenção de relatórios que fornecem estatísticas sobre o número de vezes que um comando foi executado e, sobre as variáveis que participam de um comando, o valor inicial, final, máximo e mínimo alcançados pelas variáveis.

Trata-se de uma ferramenta comercial, distribuída pela General Research Corporation, Santa Barbara, Califórnia, E.U.A.

1.2.2 TCAT

A ferramenta TCAT (Test-Coverage Analysis Tool) [REI90] realiza o teste de unidades segundo o critério teste de ramos lógicos. Este tipo de teste de ramos divide os predicados encontrados em uma condição em vários "if's", onde cada

"if" só contém um dos predicados Esta ferramenta insere código adicional (além do de monitoração) para "cobrir" esses ramos lógicos.

TCAT produz estatísticas da cobertura dos ramos de cada módulo em análise, para o último caso de teste ou para todos os casos de teste. Os relatórios que ela fornece mostram:

(1) segmentos (blocos) não executados;

(2) segmentos executados;

(3) histogramas;

(4) ganhos por caso de teste;

(5) eficiência do teste.

O usuário pode selecionar os relatórios por módulo. Existem versões disponíveis para Ada, C, COBOL, FORTRAN-77 e Pascal.

É uma ferramenta comercial fornecida por Software Research Corporation, San Francisco, Califórnia, E.U.A.

1.2.3 TCAT-Path

A ferramenta TCAT-Path [REI90] (Path-test Coverage Analysis System) faz a análise de cobertura de "todos" os caminhos de uma unidade. Para quantificar os caminhos de uma unidade é utilizado um utilitário chamado APG que lista todos os "possíveis" caminhos de cada unidade. Ele quantifica os laços através da criação de classes de equivalência para grupos de segmentos (blocos, nós) repetidos. O sistema APG pode manipular laços múltiplos, laços dentro de laços, e laços com múltipla entrada/ múltipla saída.

TCAT-Path inclui um analisador para a complexidade ciclomática de McCabe [MCC76] através da avaliação direta do grafo de fluxo de controle.

TCAT-Path instrumenta o programa fonte para testá-lo e então compila, liga e executa-o com os casos de teste para coletar dados da cobertura e analisá-los. Em seguida, o analisador de cobertura de TCAT mede a cobertura de caminhos executados relativo ao conjunto de caminhos previstos, para cada caso de teste ou conjunto de casos de testes.

TCAT-Path é uma ferramenta distribuída pela mesma empresa que distribui TCAT.

1.2.4 SCORE

Esta ferramenta suporta o teste de ramos de programas escritos em Pascal e foi implementada por Chusho [CHU87] na Hitachi, Kawasaki, Japão.

Ela implementa dois tipos de teste de ramos baseados em duas medidas de cobertura distintas. A primeira medida de cobertura é baseada na forma tradicional que consiste no quociente entre o número de ramos executados e o número total de ramos. A segunda medida baseia-se na aplicação do conceito de arco primitivo e consiste no quociente entre o número de ramos (arcos) primitivos executados e o número total de ramos (arcos) primitivos.

Chusho argumenta, e mostra experimentalmente, que essa segunda medida contribui para uma melhor análise de cobertura, pois evita a consideração de arcos não essenciais na medida de cobertura. Ainda, monitorando somente os arcos primitivos, tem-se uma redução de 40% no número de arcos que precisam ser monitorados.

1.2.5 Ferramenta de Herman [HER76]

Esta ferramenta suporta a aplicação do primeiro critério baseado em análise de fluxo de dados. O critério proposto por Herman é semelhante ao critérios *todos-c-usos* proposto por Rapps e Weyuker [RAP82, RAP85] seis anos depois.

A ferramenta faz análise de cobertura para o critério todos-c-usos de programas escritos em FORTRAN. Essa ferramenta possui algumas características interessantes, pois tenta tratar aspectos inter-procedurais e associações definição-uso de elementos de vetores e matrizes. O tratamento do fluxo de dados inter-procedural é rudimentar visto que implica em o usuário indicar quais os parâmetros de uma sub-rotina que são definidos quando essa sub-rotina é chamada na unidade em teste. As associações entre a definição de um elemento de um vetor e seu uso são tratadas como se cada elemento de um vetor fosse uma variável distinta; para monitorar essas "novas" variáveis, a ferramenta instrumenta o número do elemento do vetor sempre que este é referenciado no código fonte.

1.2.6 ASSET

ASSET é a ferramenta desenvolvida por Frankl e Weyuker [FRA87]; suporta basicamente a aplicação dos critérios de Rapps e Weyuker [RAP82, RAP85] em programas escritos em Pascal.

A ASSET não faz nenhuma consideração sobre as variáveis passadas por referência em chamadas de procedimento e função, ou seja, não faz nenhum tratamento inter-procedural. ASSET possui uma interface gráfica que apresenta o grafo de fluxo de controle e provê uma interação amigável com o usuário.

Essa ferramenta ainda fornece geração automática do programa executável mas não fornece os programas "drivers" e "stubs" eventualmente necessários. Aparentemente, não é provida nenhuma forma de gerenciamento dos dados de teste (entradas, saídas) gerados na sessão de trabalho.

1.2.7 Ferramenta de Korel e Laski [KOR85]

A ferramenta de Laski e Korel [KOR85] suporta a aplicação do critério *contexto de dados (ordenado)* [LAS83]. É uma ferramenta muito simples que fornece os *contextos de dados* exigidos pelo critério e faz a análise de cobertura de um conjunto de casos de teste.

A ferramenta possui uma interface com o usuário bem pobre, sem a possibilidade de visualizar o grafo de fluxo de controle e sem mecanismos de gerenciamento dos dados de teste. Trata-se de uma ferramenta monolinguagem, pois suporta somente o teste de programas em Pascal.

Aparentemente, este protótipo visa apenas demostrar que o critério pode ser automatizado.

1.2.8 PROTESTE

PROTESTE [PRI90] tem como objetivo um ambiente completo para suporte ao teste estrutural de programas, incluindo tanto critérios baseados unicamente no fluxo de controle (e.g.: critérios todos-nós e todos-ramos) como critérios baseados em análise de fluxo de dados (e.g.: critérios de Rapps e Weyuker e Potenciais Usos).

Esse ambiente inclui uma interface gráfica que permite visualizar o grafo de fluxo de controle e permite que o usuário tenha uma interação amigável com o ambiente.

PROTESTE possui um processador simbólico que permite ao usuário determinar os dados de entrada que executam um particular caminho. Dessa maneira, PROTESTE é uma ferramenta que auxilia na derivação dos dados de entrada para um caso de teste.

PROTESTE suporta o teste de programas escritos em Pascal; porém, o ambiente pode ser ajustado para outras linguagens através da utilização de uma ferramenta que gera analisadores de código fonte específicos para cada linguagem.

O ambiente faz análise de cobertura dos critérios suportados, implicando a criação de uma versão instrumentada da unidade em teste que deve ser compilada e ligada com os eventuais programas "drivers" e "stubs". Na versão de PROTESTE descrita em [PRI90] não ocorre a geração automática de "drivers" e "stubs" e do programa exeucutável; entretanto, é intenção prover essas facilidades.

O ambiente PROTESTE é um protótipo desenvolvido na Universidade Federal do Rio Grande do Sul.

1.2.9 Test Inc

Test Inc [LUT90] realiza o teste baseado em fluxo de dados (critério todos-c-usos) tanto dentro dos limites quanto fora dos limites do procedimento em análise, ou

seja, ela considera aspectos intra- e extra-procedurais. Mais ainda, ela faz uma análise incremental da unidade em teste verificando os pares definição-uso que foram alterados, ou foram afetados pelas alterações realizadas, e verifica aqueles casos de teste que não implica em ganhos de cobertura do teste.

O teste de fluxo de dados através dos limites do procedimento requer informações sobre definições e usos que extrapolam o procedimento, incluindo definições de variáveis globais, parâmetros formais que alcançam o fim do procedimento e parâmetros reais que alcançam a chamada de procedimento. Para poder monitorar a execução dos pares definições-usos inter-procedurais, Test Inc monitora as amarrações de variáveis nas chamadas de procedimentos, registrando a amarração entre o parâmetro formal e o parâmetro real e os valores retornados na saída. Essas amarrações são utilizadas para renomear as variáveis quando da avaliação da execução de um par definição-uso.

Test Inc possui uma *história* dos casos de testes que associa os pares definições-usos com os casos de testes utilizados para testá-los. Em resposta a uma mudança em um programa previamente testado, Test Inc verifica quais pares foram alterados, atualiza a história, e determina os pares a serem re-testados.

Test Inc faz uma análise da história dos casos de teste para determinar um conjunto mínimo de casos de teste que satisfaça o teste de fluxo de dados. O usuário é quem decide se elimina ou não os casos de teste redundantes.

Test Inc é um protótipo de pesquisa desenvolvido na University of Pittsburgh.

1.2.10 Comparando as Ferramentas

As ferramentas comerciais de teste estrutural de software (RXVP80, TCAT e TCAT-Path) representam o *estado da prática*. Assim, como as ferramentas comerciais suportam a aplicação de critérios estruturais baseados somente no grafo de fluxo de controle (todos-caminhos, todos-ramos e todos-nós), o que se tem na prática é a utilização desse critérios de teste na indústria.

Uma decorrência direta do fato dessas ferramentas serem comerciais é que elas são mais sofisticadas em termos de relatórios fornecidos e interface com o usuário. Outra característica de sofisticação é que elas permitem ao usuário fazer a análise de cobertura de várias unidades simultaneamente. Se por um lado essa característica possibilita testar várias unidades concomitantemente, por outro, impede que certos caminhos (que são executados somente através do teste isolado da unidade) sejam testados. O teste isolado da unidade implica a construção de unidades "drivers" e "stubs"; o ideal seria que as ferramentas fornecessem para o usuário um arcabouço dessas unidades; entretanto, nenhuma das ferramentas de teste comerciais descritas aqui têm essa capacidade.

As ferramentas comerciais são todas voltadas para uma única linguagem de programação, apesar de possuírem versões para uma série de linguagens. Essa

característica monolinguagem pode ser entendida como uma tentativa de tornar as ferramentas mais eficientes na fase de análise estática do código fonte ou como um estratégia de vendas. A experiência com o uso de algoritmos configuráveis para a realização da análise estática tem mostrado que esses algoritmos não são um gargalo no desempenho das ferramentas [MAL91a].

As ferramentas de teste orientadas para o teste com critérios baseados em análise de fluxo de dados implementam, com exceção de PROTESTE, somente uma família de critérios. Essas ferramentas diferem na abordagem com que são determinados os requisitos de teste e nas características encontradas em cada uma quanto à interface com o usuário, facilidades para determinação dos dados de entrada, facilidades para tratar várias linguagens e gerenciamento da sessão de trabalho.

ASSET, a ferramenta de Laski e Korel e PROTESTE não fazem considerações sobre o fluxo de dados inter-procedural da unidade em teste para determinação das associações requeridas pelos critérios. Já a ferramenta de Herman, a POKE-TOOL e, principalmente, Test Inc realizam algum tipo de tratamento inter-procedural. Herman utiliza um mecanismo rudimentar, pois exige intervenção do usuário; a POKE-TOOL trata os aspectos inter-procedurais detectando as variáveis que podem ser definidas por referência na chamada de um procedimento e assumindo uma estratégia de pior caso (ver Seção 2.3.4); e Test Inc faz uma análise mais exaustiva, até mesmo monitorando as amarrações ocorridas entre os parâmetros formais e reais durante a execução de um caso de teste.

A interface com o usuário das ferramentas baseadas em fluxo de dados variam de acordo com o aspecto que foi encarado como o mais importante. Por exemplo, PROTESTE possui uma interface amigável, com facilidades para visualizar o grafo de fluxo de controle, mas não provê mecanismos básicos para gerenciamento dos dados de teste (e.g.: entradas, saídas e caminhos) obtidos durante a execução dos casos de teste; o mesmo acontece com ASSET. As ferramentas de Herman e Laski e Korel possuem uma interface muito simples sem possibilidade de visualizar o grafo de fluxo de controle e sem gerenciamento dos dados de teste gerados. A POKE-TOOL também possui uma interface simples mas permite que seja criada uma base de dados com as informações relevantes geradas na sessão de trabalho. Test Inc também não possui essas capacidades, mas faz um tratamento interessante dos casos de teste com a criação da *história* dos casos de teste.

Das ferramentas discutidas acima, apenas PROTESTE fornece subsídios para a determinação dos dados de entrada necessários para exercitar um particular caminho. As demais ferramentas fornecem somente os requisitos de teste exigidos pelos critérios de teste suportados. Entretanto, o mecanismo de execução simbólica de programas de PROTESTE pode levar a um conjunto muito grande de equações que representam o caminho, o que torna necessário um simplificador para reduzir o número de equações. Esse número excessivo de equações dificulta a determinação

dos dados de entrada dos casos de teste. Aparentemente, este mecanismo de execução simbólica não está integrado a PROTESTE, como mostram os menus contidos em [PR190].

Somente PROTESTE e a POKE-TOOL fornecem mecanismos para que um usuário configurador gere uma nova configuração para o teste de programas escritos em uma nova linguagem de programação. PROTESTE utiliza uma ferramenta que gera *um programa* para análise do código fonte na nova linguagem de programação e a POKE-TOOL faz essa análise específica através de um analisador sintático dirigido *por tabelas.*

As ferramentas comerciais possuem a sofisticação adicional de possibilitar a geração do programa executável automaticamente. A única ferramenta baseada em fluxo de dados que realiza tal tarefa é ASSET, que necessita de um arquivo *makefile* para realizá-la. Uma facilidade interessante seria a geração de programas "drivers" e "stubs" para auxiliar o teste de unidades; entretanto, nenhuma das ferramentas estudadas possuem essa capacidade.

A ferramenta POKE-TOOL, descrita nesse trabalho, possui as características básicas das ferramentas acima: é uma ferramenta de suporte à aplicação de um conjunto de critérios fazendo basicamente análise de cobertura para um conjunto de casos de teste. Ela provê uma interface simples para o usuário (sem maiores grafismos) mas fornece uma organização dos dados de teste que só é encontrada nas ferramentas comerciais. Leva em conta considerações inter-procedurais que tornam o teste com a POKE-TOOL mais exaustivo e abrangente do que o teste com a maioria das ferramentas; e permite que o usuário a configure para novas linguagens, que é uma característica muito interessante dada a enorme gama de linguagens em uso hoje em dia. A forma utilizada para determinar o requisitos de teste dos critérios PU e o mecanismo de avaliação dos casos de teste utilizado na POKE-TOOL tornam facilitada a tarefa de inserir novos critérios de teste. Adicionalmente, a POKE-TOOL, juntamente com ASSET, é a única ferramenta baseada em fluxo de dados utilizada para o teste de programas reais [WEY90, MAL91a].

Nos capítulos que se seguirão, a POKE-TOOL será apresentada mais detalhadamente quanto à sua implementação e utilização.

1.3 Organização da Tese

No Capítulo 2 são introduzidos conceitos básicos referentes às atividades de teste estrutural, os aspectos teóricos que nortearam a automatização dos critérios Potenciais Usos e a Arquitetura da ferramenta POKE-TOOL. O projeto detalhado da POKE-TOOL e os seus módulos constituintes, que implementam as funções discutidas no Capítulo 2, são descritos detalhadamente no Capítulo 3. Aspectos de configuração da POKE-TOOL para uma linguagem de programação específica,

a configuração para a linguagem C e um exemplo de utilização são apresentados no Capítulo 4. As conclusões são apresentadas no Capítulo 5.

Capítulo 2

Arquitetura e Aspectos de Implementação da POKE-TOOL

Neste capítulo apresentam-se a arquitetura e os principais aspectos teóricos de implementação de uma ferramenta de teste - denominada POKE-TOOL (**PO**tential Uses **CRI**teria **TOOL** for program testing) - que apóia a utilização dos *Critérios Potenciais Usos* (PU) [MAL88a, MAL88b, MAL89]. Os critérios Potenciais Usos são critérios de teste estrutural, baseados na análise de fluxo de dados e consistem, fundamentalmente, em variações da família de critérios apresentada por Rapps e Weyuker [RAP82, RAP85]; são denominados: critérios *todos-potenciais-du-caminhos*, *todos-potenciais-usos* e *todos-potenciais-usos/du*. No contexto de teste de software, métodos e critérios de projeto de casos de teste baseados em análise de fluxo de dados requerem que as interações que envolvem definições de variáveis de programa e subsequentes referências afetadas por essas definições sejam testadas; por sua vez, os Critérios Potenciais Usos requerem associações que implicam no exercício de caminhos entre a definição de uma variável e um possível (potencial) uso desta variável. Analogamente aos outros critérios, os critérios Potenciais Usos necessitam de uma ferramenta para sua aplicação efetiva.

Com a implementação da ferramenta POKE-TOOL pretende-se, além de auxiliar o uso prático dos critérios Potenciais Usos, viabilizar a realização de comparações entre estes critérios e os demais critérios de teste estrutural, bem como a avaliação da adequação destes critérios a classes de erros.

2.1 Conceitos Básicos

A terminologia e conceitos apresentados nesta seção são essencialmente uma síntese dos trabalhos de Rapps, Frankl e Weyuker [RAP82, WEY84, RAP85, FRA85, FRA88] acrescidas de alguns conceitos e terminologia pertinentes aos critérios Potenciais Usos [MAL91a, MAL91b, MAL91c].

Considerando um grafo de fluxo de controle $G(N,A,s)$, onde N é o conjunto de nós, A o conjunto de arcos e s o nó inicial, define-se: *caminho* como sendo uma sequência finita de nós $(n_i,\ldots,n_k)$, $k \geq 2$, tal que existe um arco de n_i para n_{i+1}, $i = 1,2,\ldots,k-1$; *caminho simples* como um caminho onde todos nós, exceto possivelmente o primeiro e o último, sejam distintos; *caminho livre de laço* como um caminho com todos os nós distintos; e *caminho completo* onde o nó inicial é um nó de entrada e o nó final é um nó de saída do grafo G.

As ocorrências de uma variável em um programa podem ser uma *definição de variável*, um *uso de variável* ou uma *indefinição*.

Conforme o modelo de dados definido na Seção 2.3.4, uma *definição de variável* ocorre quando um valor é armazenado em uma posição de memória. Em geral, em um programa, uma ocorrência de variável é uma definição se ela está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; ou iii) em chamadas de procedimentos como parâmetro de saída. A passagem de valores entre procedimentos através da passagem de parâmetros pode ser por: *valor*, *referência* ou por *nome* [GHE87]. Se a variável for passada por referência ou por nome considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são distinguidas das demais e são ditas *definidas por referência*.

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

Uma variável está *indefinida* quando, ou seu valor se torna inacessível, ou sua localização deixa de estar amarrada na memória.

O grafo de fluxo de controle estendido pela associação a cada nó i do conjunto de variáveis definidas em i ($defg(i)$) é denominado *grafo def*.

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

Um caminho livre de definição $(n_1,n_2,\ldots,n_j,n_k)$ onde o caminho $(n_1,n_2,\ldots,n_j)$ é um caminho livre de laço e n_1 tem uma definição de x, é denominado *potencial-du-caminho c.r.a x*.

Para a definição dos critérios Potenciais Usos é necessária a introdução dos seguintes conceitos. $pdcu(x,i)$ = { nós j | existe um caminho livre de definição c.r.a x de i a j}; $pdpu(x,i)$ = { arcos (j,k) | existe um caminho livre de definição c.r.a x de i a (j,k)}; uma *potencial associação definição-c-uso* é a tripla $[i,j,x]$ *onde* $x \in defg(i)$ e $j \in pdcu(x,i)$; e uma *potencial associação definição-p-uso* é a tripla $[i,(j,k),x]$ *onde* $x \in defg(i)$ e $(j,k) \in pdpu(x,i)$. Uma *potencial associação* é definida como uma potencial associação definição-c-uso, uma associação definição-p-uso ou

um potencial-du-caminho.

Um caminho $\pi_1 = (i_1, \ldots, i_k)$ é dito estar incluído em um conjunto Π de caminhos se Π contém um caminho $\pi_2 = (n_1, n_2, \ldots, n_m)$ tal que $i_1 = n_j, i_2 = n_{j+1}, \ldots, i_k = n_{j+k-1}$, para algum j, $1 \leq j \geq m - k + 1$. Então, π_1 está incluído em π_2 ou que π_1 é um sub-caminho de π_2.

Um caminho completo π cobre um potencial associação definição-c-uso */i,j,x/* (respectivamente, uma potencial associação definição-p-uso */i,(j,k),x/*) se ele inclui um caminho livre de definição c.r.a x de i para j (respectivamente, de i para *(j,k)*). π cobre um potencial-du-caminho π_1 se π_1 está incluído em π. Um conjunto Π de caminhos cobre uma potencial associação se algum elemento do conjunto cobrir.

2.1.1 Critérios Potenciais Usos

Os critérios potenciais usos – todos-potenciais-usos, todos-potenciais-usos/du e todos-potenciais-du-caminhos – requerem basicamente que caminhos livres de definição, em relação a qualquer nó i que possua definição de variável e a qualquer variável x definida em i, sejam executados, independentemente de ocorrer uso dessa variável nesses caminhos. Neste sentido, pode-se verificar, por exemplo, que o valor de x não foi alterado nesses caminhos (possivelmente devido a efeitos colaterais) ganhando-se, desta forma, maior confiança de que a computação correta é realizada; isto vem de encontro à filosofia discutida por Myers [MYE79]: um erro está claramente presente se um programa não faz o que supõe-se que ele faça, mas erros estão também presentes se um programa faz o que supõe-se que não faça. Seja Π um conjunto de caminhos completos:

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

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

Critério Todos-potenciais-du-caminhos - Π satisfaz o critério todos-potenciais-du-caminhos se para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui todos *potenciais du-caminhos* c.r.a x em relação ao nó i[1].

2.2 Arquitetura da Ferramenta POKE-TOOL

Nesta seção é discutida a arquitetura proposta para a POKE-TOOL e são abordadas as funções suportadas pela ferramenta.

Segundo Deutsch [DEU82], existem 5 funções básicas que devem ser realizadas por uma ferramenta deste tipo:

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

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

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

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

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

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

Semelhantemente à ferramenta de teste implementada por Frankl e Weyuker ASSET [FRA85] -, a POKE-TOOL [MAL89][2], cuja arquitetura geral está ilustrada na Figura 2.1, terá como entrada o programa a ser testado, a seleção do critério a ser utilizado e um conjunto de casos de teste. Na hipótese do conjunto de casos de teste fornecido ser vazio, a POKE-TOOL fornecerá um conjunto de caminhos necessários para satisfazer o critério selecionado, orientando desta forma a própria seleção de casos de teste. Se o conjunto de casos de teste fornecido não for

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

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

vazio, será produzida uma relação de caminhos requeridos pelo critério mas ainda não executados. Pretende-se dar suporte para que esta ferramenta aceite programas implementados em linguagens de programação distintas, tais como PASCAL, C, COBOL, etc. Este fato exige um pré-processamento do programa fonte de forma que as particularidades de cada uma das linguagens sejam tratadas uniformemente.

Como pode ser extraído da Figura 2.1, a POKE-TOOL é constituída basicamente de nove funções descritas abaixo:

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

CÁLCULO DOS ARCOS PRIMITIVOS: calcula os arcos primitivos (definidos na Seção 2.3.5) do grafo de fluxo de controle. O conjunto de arcos primitivos servirá de base para a instrumentação do programa fonte e para a construção dos autômatos utilizados na avaliação da adequação de conjuntos de casos de teste. O algoritmo utilizado para determinação dos arcos primitivos, descrito neste relatório, é uma variante do algoritmo de Chusho [CHU87]. Esta variante consiste numa adequação desse algoritmo para a utilização do conceito de arcos primitivos no contexto de testes estruturais baseados em análise de fluxo de dados.

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

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

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

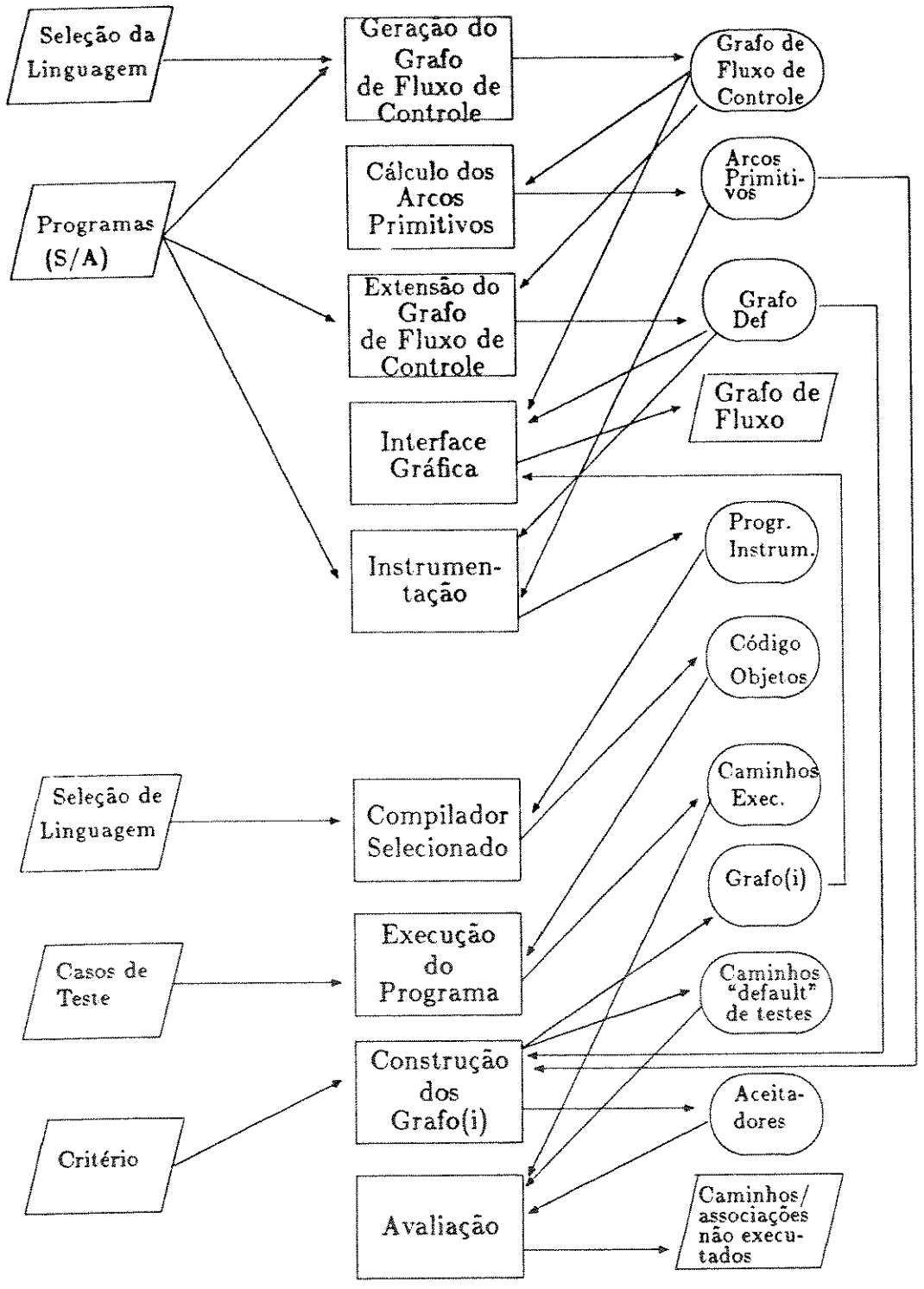


Figura 2.1: Arquitetura da Ferramenta de Teste Estrutural POKE-TOOL.

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

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

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

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

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

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

Entre os caminhos requeridos pode existir um caminho não executável; a ferramenta POKE-TOOL, na sua primeira versão, não dá nenhum suporte para determinação da executabilidade de um determinado caminho - o que é uma questão indecidível -; métodos heurísticos foram desenvolvidos para tratar esse problema [FRA85, FRA88].

Observe-se que a ferramenta POKE-TOOL pode constituir-se também num suporte extremamente útil tanto às atividades de depuração como às de manutenção de programas.

A seguir são introduzidos aspectos teóricos que nortearam a automatização dos critérios PU; ainda, os termos técnicos e conceitos, introduzidos informalmente na explanação acima, são definidos precisamente.

2.3 Aspectos Teóricos de Implementação

Nas seções seguintes, são apresentados os aspectos teóricos que nortearam a implementação da POKE-TOOL, a saber: a Linguagem Intermediária (LI) [CAR91]; o Modelo de Fluxo de Programa; o Modelo de Instrumentação; o Modelo de Fluxo de Dados; utilização do conceito de arcos primitivos [CHU87] no contexto de teste estrutural baseado em fluxo de dados e o Modelo de Descrição dos elementos — caminhos ou associações — requeridos pelos critérios Potenciais Usos. Deve-se salientar que estes aspectos devem ser considerados para qualquer linguagem em que a POKE-TOOL venha a ser configurada. As Seções 2.3.2, 2.3.3, 2.3.4, 2.3.5 e 2.3.6 foram extraídas e sintetizadas de [MAL91a] e incluídas neste texto para efeito de clareza e completude.

2.3.1 A Linguagem Intermediária (LI)

A linguagem intermediária (LI) tem como função principal identificar o fluxo de execução em um programa. Por isso, basicamente, se tem dois tipos de comandos na LI: comandos *sequenciais* e comandos de *controle de fluxo*. Os comandos *sequenciais* da LI indicam os comandos das linguagens procedurais que representam uma declaração de variável ou uma computação (comandos de atribuição ou chamadas de procedimentos) e que, portanto, não alteram o fluxo de execução. Os comandos de *controle de fluxo* da LI são equivalentes aos comandos das linguagens procedurais que causam *seleção*, *seleção múltipla*, *iteração* e *transferência incondicional*. Porém, antes de apresentar os comandos da LI, será apresentada a estrutura dos átomos da LI.

Uma característica própria da LI é que todos os *átomos* da linguagem são seguidos por números que identificam, respectivamente, o início do átomo no arquivo fonte da unidade em teste (a quantos bytes do começo do arquivo se inicia o átomo), o comprimento do átomo (quantos bytes tem o átomo) e a linha onde está o átomo. Dessa maneira, utilizando a notação de Backus-Naur, um átomo da LI teria a seguinte estrutura:

$< atm\_li > ::= < atomo >< inicio >< comprimento >< linha >.$

Onde

$$< atomo > ::= \$DCL|\$S|\$IF|\$CASE|\$ROTC|\$ROTD|\$WHILE|\$FOR|\{|\}|\$C|\$NC|\$REPEAT|\$UNTIL|\$GOTO|LABEL|\$BREAK|\$CONTINUE|\$RETURN|\$CC|\$ELSE$$

e

$< inicio > ::= NUM,$
$< comprimento > ::= NUM$ e
$< linha > ::= NUM.$

Os terminais acima representam a seqüência de caracteres indicada pelos próprios terminais; com exceção de $\$S$ que indica a sequência de caracteres $\$Sd^n$ onde d pertence a $\{0, 1, \ldots, 9\}$ e $n > 0$, $\$C$ que indica a seqüência $\$C(d^n)d^n$, $\$NC$ que indica a seqüência $\$NC(d^n)d^n$, NUM que indica a seqüência d^n e $LABEL$ que indica uma seqüência de letras e caracteres sempre começando por um caractere.

A utilidade desses ponteiros dos átomos é possibilitar o acesso ao código fonte associado ao átomo da LI, o que vai ser necessário no módulo **pokernel** (ver Seção 3.6). Por exemplo, o comando na linguagem Pascal

```
var i,j,k: integer;
```

seria traduzido para a LI como

```
$DCL 150 19 10.
```

O comando LI significa uma declaração que começa a 150 bytes do início do arquivo fonte, tem 19 bytes de comprimento e está localizado na linha 10. Note-se que toda uma linha de comando em Pascal originou um átomo da LI.

Na convenção de Backus-Naur [SEB89] adotada aqui, os terminais serão representados em itálico, os não-terminais entre "<" e ">" e os meta-símbolos sublinhados.

A seguir, serão apresentados os diversos comandos da LI e para descrevê-los será utilizada notação de Backus-Naur.

Os comandos *seqüênciais* são os seguintes:

$< dcl > ::= \$DCL < inicio >< comprimento >< linha >$ e
$< s > ::= \$S < inicio >< comprimento >< linha >.$

Onde $< dcl >$ denota uma *declaração de variável* e $< s >$ uma computação; sendo que uma computação pode ser uma atribuição de valor a uma variável (através de uma expressão ou chamada de função), ou somente uma chamada de procedimento.

Por exemplo, o comando em C

```
x = fopen("arquivo.c","r");
```

será traduzido para

```
$S1       1024 27 23.
```

Os comandos de controle de fluxo são os seguintes:

(1) Seleção:

$$< if > ::= < if\_atm >< cond\_atm >< statement_1 > \mid$$
$$< if\_atm >< cond\_atm >< statement_1 >< else\_atm >< statement_2 >$$

onde

$< if\_atm > ::= \$IF < inicio >< comprimento >< linha >,$
$< cond\_atm > ::= \$C < inicio >< comprimento >< linha >,$
$< else\_atm > ::= \$ELSE < inicio >< comprimento >< linha >$ e

$< statement_i >$ é o não-terminal que denota todos os possíveis comandos da LI, agrupados ou não; $< statement_i >$ será definido formalmente mais adiante. Este comando significa o "if" tradicional das linguagens do estilo ALGOL, ou seja, os "comandos" em $< statement_1 >$ serão executados se $< cond\_atm >$ for verdadeiro e os "comandos" em $< statement_2 >$ serão executados caso contrário. Considere o seguinte trecho de programa em Pascal.

⋮

if j <> 0 then k:=k/j; else writeln('error: division by zero');

⋮

Este comando seria traduzido para a LI como:

⋮

```
$IF          1031     2     10
$C(1)1       1034     6     10
$S1          1046     7     10
$ELSE        1054     4     10
$S2          1059    35     10
```

⋮

A solução para a ambigüidade que poderia ser gerada pelo encadeamento de *$IF* e *$ELSE* sem delimitadores de bloco { e } foi tomada inspirada na maioria das linguagens do estilo ALGOL, onde o *$ELSE* é associado com o mais recente *$IF* sem *$ELSE*, salvo o uso explícito de chaves que podem forçar a associação apropriada. Observe-se que *$S1* acima representa o primeiro comando sequencial do programa em LI, ou seja, o número que segue os caracteres "$S" indica a ordem em que aparece o comando sequencial no programa. Os números que aparecem em *$C(1)1* indicam, respectivamente, o número de predicados que possui a condição e a ordem de aparição.

(2) Seleção Múltipla:

$< case > ::=$
$< case\_atm >< case\_cond\_atm > \{ \{ (< rotc\_atm > | < rotd\_atm >) \{ < statement > \} \}\}$
onde
$< case\_atm > ::= \$CASE < inicio >< comprimento >< linha >,$
$< case\_cond\_atm > ::= \$CC < inicio >< comprimento >< linha >,$
$< rotc\_atm > ::= \$ROTC < inicio >< comprimento >< linha >$ e
$< rotd\_atm > ::= \$ROTD < inicio >< comprimento >< linha >.$

O não-terminal < *case_atm* > representa o átomo que inicia o comando de seleção múltipla, < *case_cond_atm* > representa a condição do comando e < *rotc_atm* > representa os possíveis rótulos para as seqüências de comandos indicadas por < *statement* >. O não-terminal < *rotd_atm* > representa o rótulo para a seqüência de comandos a ser executada quando a condição não combina com nenhum rótulo < *rotc_atm* >.

A semântica do comando acima é equivalente ao comando "switch" da linguagem C, isto é, a execução começa no rótulo que ocorreu a combinação e executa os comandos desse rótulo mais os comandos dos rótulos que o seguem, a menos que seja encontrado um comando do tipo "break", que causa a imediata saída do comando de seleção múltipla, ou um comando de desvio incondicional. Esses comandos serão discutidos mais adiante.

Considere o seguinte trecho de program em Ada [GHE87]:

```
case OPERATOR of
     when "." => RESULT := OPERAND1 and OPERAND2;
     when "+" => RESULT := OPERAND1 or OPERAND2;
     when "=" => RESULT := OPERAND1 = OPERAND2;
     when others => ... produce error message ...
end case;.
```

Seria traduzido para a LI como:

```
$CASE     1001    4    5
$CC       1006    8    5
{         1016    2    5
$ROTC     1021   11    6
$S1       1033   32    6
$BREAK       0    0    0
$ROTC     1068   11    7
$S2       1081   31    7
$BREAK       0    0    0
$ROTC     1115   11    8
$S3       1127   30    8
$BREAK       0    0    0
$ROTD     1160   14    9
$S4       1175   29    9
}         1206   10   10
```

Observe-se que foi acrescentado o comando "break" no trecho em LI para manter a semântica do "case" da linguagem Ada. Note-se também que os ponteiros

do comando "break" são iguais a 0 para indicar que não há correspondência no código fonte para o comando LI em questão.

(3) Iteração

A LI fornece comandos para iteração tanto para um número fixo de repetições quanto para um número de repetições que depende de uma condição.

No caso de um número fixo de repetições, tem-se um comando semelhante ao "for" das linguagens do estilo ALGOL. O "for" da LI é definido como

$< for > ::= < for\_atm >< s_1 >< cond\_for\_atm >< s_2 >< statement >$
onde
$< for\_atm > ::= \$FOR < inicio >< comprimento >< linha >$ e
$< cond\_for\_atm > ::= \$C < inicio >< comprimento >< linha >$.

O não-terminal $< for\_atm >$ indica o comando "for" da LI, o não-terminal $< s_1 >$ representa a iniciação das variáveis de controle do "for" através de um comando sequencial, $< cond\_for\_atm >$ representa a condição, $< s_2 >$ representa o comando sequencial que altera as variáveis de controle a cada iteração do "for" e $< statement >$ representa o corpo do comando. O comando "for" da LI é inspirado no comando equivalente da linguagem C, possuindo a mesma semântica. No trecho de programa em C

```
for (i=0; i < nfiles; i++)
    fstat[i] = 0;
```

a tradução para a LI seria a seguinte.

```
$FOR      110    3    12
$S1       115    4    12
$C(1)1    120   11    12
$S2       132    3    12
$S3       139   13    13
```

A associação dos comandos do código fonte para a LI não é sempre direta como no exemplo anterior. Considere o trecho de programa em Pascal [GHE87]:

```
type day = (sunday,monday,tuesday,wednesday,thursday,friday,saturday);
var week_day: day;
...
    for week_day := monday to friday do
        ...;
```

seria traduzido para

```
$DCL       51     70     2
$DCL      124     18     3
                ...
$FOR      303      3     7
$S1       307     18     7
$C(1)1      0      0     0
$S2       326      9     7
                ...
```

Observe-se que na condição do comando "for" os ponteiros são iguais a 0, novamente indicando que o átomo da LI não possui correspondência no arquivo fonte. Obviamente, a decisão de como mapear a linguagem da unidade para a LI é do usúario configurador da POKE-TOOL.

Os comandos de iteração cujo número de repetições é dirigido por uma condição são também equivalentes aos tradicionais "while" e "repeat-until" das linguagens do estilo ALGOL. O "while" da LI é definido como

$< while > ::= < while\_atm >< cond\_while\_atm >< statement >$
onde
$< while\_atm > ::= \$WHILE < inicio >< comprimento >< linha >$ e
$< cond\_while\_atm > ::= \$C < inicio >< comprimento >< linha >$.

$< while\_atm >$ indica o comando "while", $< cond\_while\_atm >$ a condição e $< statement >$ o corpo do "while". O corpo será executado enquanto a condição em $< cond\_while\_atm >$ permanecer verdadeira.

O "repeat-until" da LI é definido como

$< repeat\_until > ::=$
$< repeat\_atm >< statement >< until\_atm >< cond\_until\_atm >$,
onde
$< repeat\_atm > ::= \$REPEAT < inicio >< comprimento >< linha >$,
$< until\_atm > ::= \$UNTIL < inicio >< comprimento >< linha >$ e
$< cond\_until\_atm > ::= (\$C|\$NC) < inicio >< comprimento >< linha >$.

$< repeat\_atm >$ indica o início do comando "repeat-until", $< statement >$ o corpo do comando, $< until\_atm >$ indica o fim do comando e $< cond\_until\_atm >$ representa a condição de término. A semântica desse comando da LI é igual ao comando equivalente das linguagens "ALGOL-like", ou seja, o corpo da iteração

é executado pelo menos uma vez e o teste da condição é realizado depois da execução do corpo. Note-se que a condição de término é composta por *$C* ou *$NC*; isto ocorre porque, na maioria das linguagens "ALGOL-like", o corpo do "repeat-until" é executado até que uma dada condição se verifique; porém, em algumas linguagens como C, o comando "repeat-until" equivalente funciona de maneira que o corpo é executado enquanto a condição permanece verdadeira. Devido a esse fato, a condição desse comando pode ser *$C*, se for o primeiro caso, e aí estaria indicando um "repeat-until" tradicional, ou *$NC*, se for o segundo caso, e nesta situação teríamos que o laço se repete até a negação da condição. Considere o seguinte trecho de programa em C [KER78]:

```
⋮
do { /* generate digits in reverse order */
     s[i++] = n % 10 + '0'; /* get next digit */
   } while ((n /= 10)>0); /* delete it */
⋮
```

Este trecho seria traduzido em LI para

```
   ⋮
$REPEAT     1000     2    12
{           1003     1    12
$S1         1042    22    13
}           1080     1    14
$UNTIL      1080     5    14
$NC(1)1     1086    14    14
   ⋮
```

(4) Desvios Incondicionais

Os comandos de desvios incondicional provocam a mudança do fluxo de execução em um programa. A LI possui um comando de transferência incondicional irrestrito do tipo "goto" e comandos de transferência incondicional controlada; estes últimos têm sua utilização limitada a algumas situações e seu efeito bem previsível.

O comando "goto" da LI é definido como

$< goto > ::= < goto\_atm >< label\_atm >$
onde
$< goto\_atm > ::= \$GOTO < inicio >< comprimento >< linha >$ e
$< label\_atm > ::= LABEL < inicio >< comprimento >< linha >$.

$< goto\_atm >$ representa o comando "goto" da LI e $< label\_atm >$ representa o rótulo para onde deve ser dirigido o fluxo de execução quando é encontrado o comando "goto".

Existem mais três comandos de desvio incondicional na LI; são eles

$< break > ::= \$BREAK < inicio >< comprimento >< linha >$,
$< continue > ::= \$CONTINUE < inicio >< comprimento >< linha >$ e
$< return > ::= \$RETURN < inicio >< comprimento >< linha >$.

Esses comandos de transferência incondicional foram inspirados nos seus homônimos da linguagem C e, por isso, possuem efeitos idênticos. O "break" causa o fim do comando de iteração ("for", "while" ou "repeat-until") mais próximo que o engloba. Ainda, dentro de um comando "case" da LI, o "break" causa o desvio para o primeiro comando fora do "case". O comando "continue" provoca o desvio para a *próxima iteração* do laço que o engloba. No caso dos comandos "repeat-until" e "while", ao encontrar-se o "continue", o fluxo de execução é desviado para o teste da condição da iteração; no caso do comando "for", o fluxo de execução é desviado para o comando que altera as variáveis de controle. O comando "return" causa o fim do procedimento que está sendo executado e o retorno para a unidade que o chamou.

Até aqui foram descritos os comandos individuais da LI; entretanto, para concluir a definição da LI, ainda falta definir como se agrupa comandos na LI e como esses comandos são organizados em um programa. O não-terminal $< statement >$, muito utilizado acima, representa um único comando da LI ou um agrupamento deles e é definido como

$$
\begin{aligned}
< statement > ::= \; & \{ \{ < statement > \} \} \mid \\
& < dcl > \mid \\
& < s > \mid \\
& < if > \mid \\
& < case > \mid \\
& < for > \mid \\
& < while > \mid \\
& < repeat\_until > \mid \\
& LABEL < statement > \mid \\
& < goto > \mid \\
& < break > \mid \\
& < continue > \mid \\
& < return >.
\end{aligned}
$$

Os programas em LI são definidos da seguinte maneira:

< *program* > ::= { (< *dcl* > | < *s* >) } { < *statement* > }.

Exemplo

A seguir, é mostrado o resultado da tradução de um programa (de Kernighan e Ritchie [KER78,pag.55]), na linguagem C, para uma versão do mesmo em **LI**.

```
main() /* count digits, white space, others */
{
  int c, i, nwhite, nother, ndigit[10];
  nwhite = nother = 0;
  for (i=0; i < 10; i++)
      ndigit[i] = 0;
  while ((c = getchar()) != EOF)
      switch (c) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
            ndigit[c-'0']++;
            break;
      case ' ':
      case '\n':
      case '\t':
            nwhite++;
            break;
      default:
            nother++;
            break;
      }
   printf("digits =");
   for (i=0; i < 10; i++)
      printf(" %d", ndigit[i]);
```

```
  printf("\nwhite space = %d, other = %d\n",
      nwhite, nother);
}
```

Arquivo obtido da tradução do exemplo acima:

```
$DCL                1      6      1
{                  51      1      2
$DCL               63     37      3
$S1               109     20      4
$FOR              138      3      5
$S2               145      4      5
$C(O1)1           150      6      5
$S3               158      3      5
$S4               178     14      6
$WHILE            201      5      7
$C(O1)2           207     24      7
$CASE             245      6      8
$CC               252      3      8
{                 257      1      8
$ROTC             272      9      9
$ROTC             295      9     10
$ROTC             318      9     11
$ROTC             341      9     12
$ROTC             364      9     13
$ROTC             387      9     14
$ROTC             410      9     15
$ROTC             433      9     16
$ROTC             456      9     17
$ROTC             479      9     18
$S5               507     16     19
$BREAK            542      6     20
$ROTC             562      9     21
$ROTC             585     10     22
$ROTC             609     10     23
$S6               638      9     24
$BREAK            666      6     25
$ROTD             686      8     26
$S7               713      9     27
$BREAK            741      6     28
}                 761      1     29
```

```
$S8                 771     19      30
$FOR                799      3      31
$S9                 804      4      31
$C(01)3             809      5      31
$S10                816      3      31
$S11                834     26      32
$S12                869     79      33
}                   952      1      35
```

Um outro exemplo é fornecido no arquivo ENTAB.LI do Apêndice A, que consiste na tradução do programa ENTAB.C para a linguagem LI.

2.3.2 Modelo de Fluxo de Controle

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

As Figuras 2.2, 2.3, 2.4, 2.5, 2.6 e 2.7 representam as construções básicas do grafo de fluxo de controle adotadas para os comandos da linguagem LI, a menos dos comandos de desvios incondicionais. Note que, para o comando "case", existe uma possível ligação do nó k para o nó l; esta ligação existe no caso em que não foi especificado um rótulo "default" no comando "case". O grafo de controle de uma unidade é obtido pela simples concatenação dessas construções básicas. Observe que o grafo de fluxo de controle da unidade em teste é independente da linguagem de implementação da unidade; no entanto, o usuário configurador da POKE-TOOL deverá levar este modelo em consideração pois, de certa forma, ele reflete aspectos da semântica da LI e terá forte influência na instrumentação da unidade em teste.

A representação dos comandos de desvios condicionais e incondicionais introduz modificações profundas no fluxo de controle e, consequentemente, no modelo e na instrumentação. O comando de desvio incondicional irrestrito *goto* encerra a caracterização de um bloco, sendo que existirá um arco do bloco que contém o

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

Os modelos correspondentes às alternativas de ocorrências dos comandos de desvio com os demais comandos da LI são representados na Figura 2.6. Observe-se que alguns nós não alcançáveis podem ser gerados, facilitando a identificação de código não executável, se existir código associado a esses nós; esses nós poderiam facilmente ser eliminados através de um pós-processamento ou mesmo pela modificação do modelo. Esses nós não introduzem nenhuma complicação para a implementação das outras funções da POKE-TOOL. Atenção especial deve ser dada na instrumentação quanto aos comandos de desvio incondicional; por exemplo, no caso de uma unidade ter mais de um comando *return*.

2.3.3 Modelo de Instrumentação

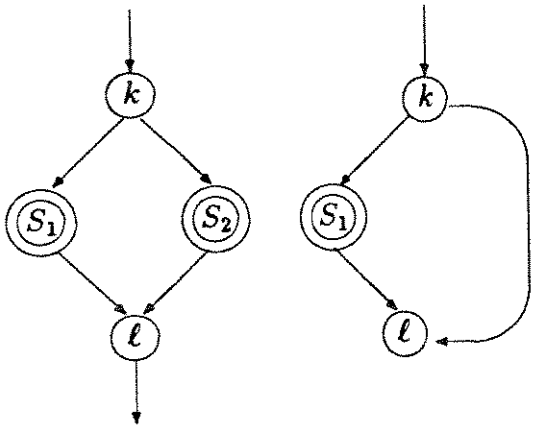
A instrumentação visa prover facilidades para a análise posterior à execução dos casos de testes, como por exemplo, a análise de adequação de um dado conjunto de casos de teste. Para tanto, este módulo modifica, com inserção de código fonte, a própria unidade em teste, gerando uma nova versão do programa,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

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

A escrita de todos os nós foi adotada para simplificar a implementação, pois, para os modelos adotados de avaliação e de descrição dos caminhos e associações

Comandos de seleção

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



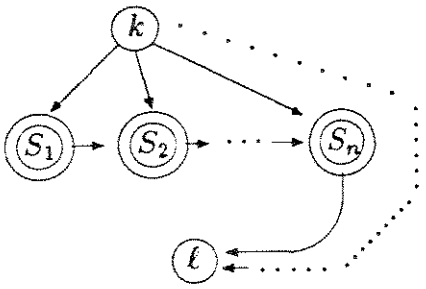
S_1 - representa o subgrafo correspondente ao < statement$_1$ >

S_2 - representa o subgrafo correspondente ao statement$_2$.

k - associado a <cond-atm>.

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

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



S_i i=1, ..., n representa os subgrafos correspondentes ao < statement$_i$ > relativos a cada uma das alternativas da seleção múltipla

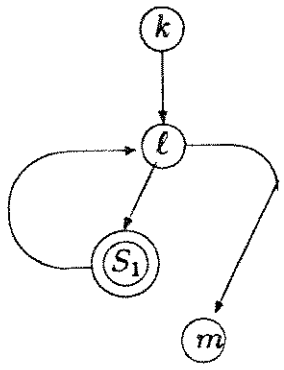
k - associado a <case-cond-atm>.

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

Figura 2.2: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos de Seleção.

Comandos de Iteração

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

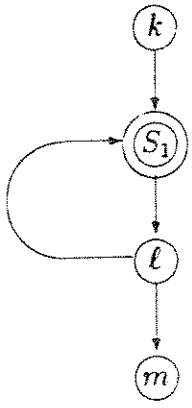


(S_1) representa o subgrafo correspondente a < *statement* >, ou seja, ao corpo do laço

(ℓ) – associado a < cond-while-atm>

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

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



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

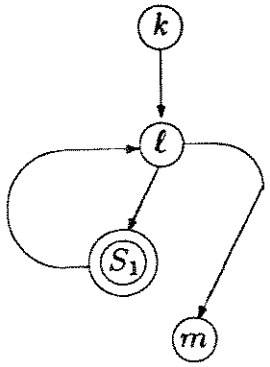
(ℓ) – associado a < cond-until-atm>

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

Figura 2.3: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos de Iteração — "while" e "repeat".

Comandos de Iteração

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



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

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

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

(ℓ) - associado a < cond-for-atm>

instrumentação:

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

Figura 2.4: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comando de Iteração "for".

Comandos Sequenciais

< dcl > e < s >

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

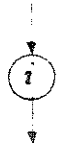
Comandos de Desvio Incondicional

< goto >

< break >

< continue >

< return >



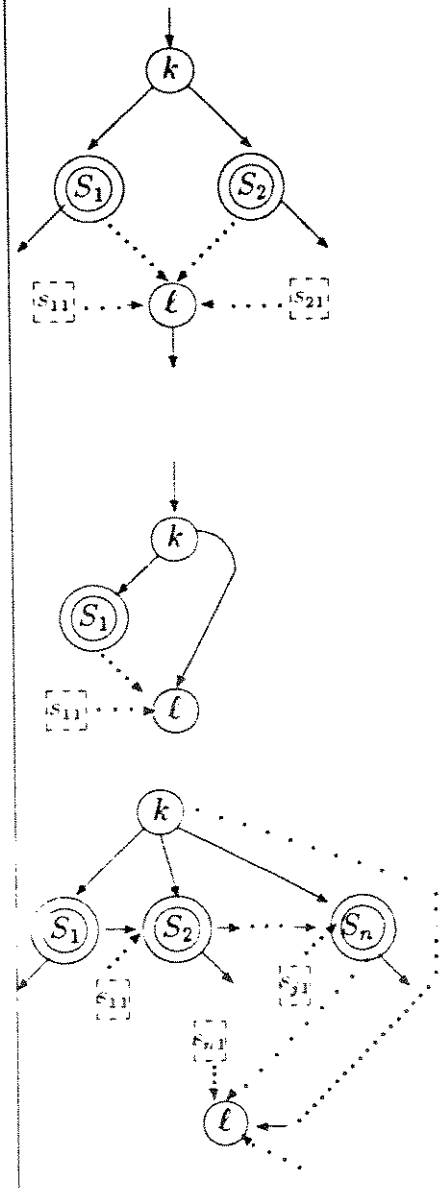
estes comandos são sempre o último comando associado ao bloco i (nó i), ou seja, todos têm a ação de encerrar um bloco; a ação de fluxo de controle destes comandos é melhor entendida e depende do local da ocorrência do comando (ver Figuras 2.6 e 2.7)

Figura 2.5: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos Sequenciais e de Desvio Incondicional.

Comandos de Seleção &
Comandos de Desvio Incondicional

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

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



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

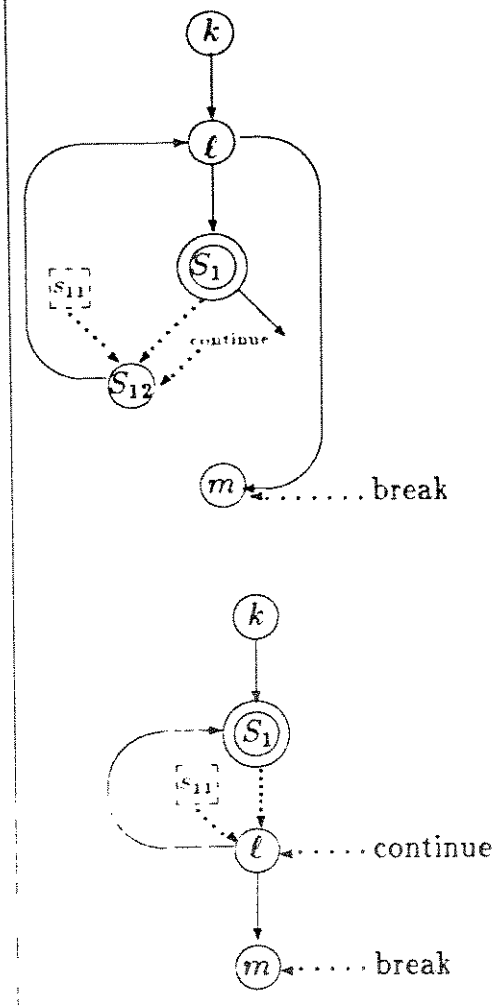
S_{i1} – representa subgrafos correspondentes a comandos após a ocorrência de comandos de desvio incondicional em <statement$_i$ >. Estes blocos podem representar comandos não alcançáveis.

instrumentação: Seja e o nó de saída da unidade em teste. Seja i o nó de entrada do subgrafo S_i e i_1 o nó de entrada do subgrafo S_{i1}. Seja d o nó em que ocorre o comando de desvio incondicional. No nó d é inserida a ponta de prova d e adicionalmente a ponta de prova e se ocorrer o comando "return". O caso em que o comando "if" está aninhado em uma estrutura "while", "for" ou "repeat" é tratado na Figura 2.7. Aos nós i e i_1 são inseridas as pontas de prova i e i_1.

Figura 2.6: Modelo de Fluxo de Controle e Instrumentação considerando os Comandos de Desvio Incondicional da LI

Comandos de Iteração &
Comandos de
Desvio Incondicional

<while> ::= < while-atm >< cond-while-atm>< statement>
<repeat-until>::=<repeat-atm><statement><until-atm><cond-until-atm>
<for> ::= < for-atm >< S_1 >< cond-for-atm>< S_2 >< statement>



S_1 – representa o subgrafo correspondente aos comandos do início de < *statement* > até a ocorrência de um comando de desvio incondicional

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

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

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

Figura 2.7: Modelo de Fluxo de Controle e Instrumentação considerando os Comandos de Desvio Incondicional da LI

requeridos, seria suficiente inserir-se pontas de prova nos nós n que contivessem definição de variáveis ou que constituissem arcos primitivos [CHU87] (ver Seção 2.3.5). Além disto, a escrita de todos os nós pode facilitar a implementação de outras funções; por exemplo, a determinação do número de vezes que um determinado comando foi executado.

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

As Figuras 2.2, 2.3, 2.4, 2.5, 2.6 e 2.7 ilustram a instrumentação associada aos comandos da LI nas suas mais diversas possibilidades. Um fato importante a ser observado é que nenhuma modificação é introduzida nos comandos originais do programa fonte, sendo que somente pontas de provas (rotuladas ou não) e comandos de desvios incondicionais para pontas de provas rotuladas são inseridas. Obviamente, são também inseridos comandos para a abertura e fechamento do arquivo que conterá o caminho executado (arquivo PATH.TES), assim como, comandos de declaração e de iniciação das variáveis necessárias para a correta implementação desta função.

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

sem distinção do nível de recursão; esta última alternativa é incompatível com o modelo de avaliação proposto.

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

2.3.4 Modelo de Fluxo de Dados

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

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

Para a correta geração do grafo def é necessário precisar o que considera-se uma definição de variável. Uma *definição de variável* ocorre quando um valor é armazenado em uma posição de memória . Em geral, uma ocorrência de variável em um programa é uma definição se ela está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; ou iii) em chamadas de procedimentos como parâmetro de saída. A passagem de valores entre procedimentos através da passagem de parâmetros pode ser por: *valor*, *referência* ou por *nome* [GHE87]. Se a variável for passada por referência ou por nome considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são distinguidas das demais e são ditas *definidas por referência*; esta distinção é utilizada na geração dos grafo(i) (ver seção 2.3.6), ou seja, na determinação dos caminhos livres de definição para as variáveis definidas em i. Resumidamente, uma variável v_1 definida por referência faz parte do conjunto de variáveis definidas em i, ou seja $v_1 \in defg(i)$, para construção dos grafo(i), mas qualquer definição por referência de v_1 em um nó $j \neq i$ não é considerada como redefinição de v_1; este enfoque é conservador no sentido de que não deixa de requerer nenhum fluxo de dados que possa existir, ou seja, a seleção de caminhos

e associações é mais rigorosa.

Uma complicação é introduzida quanto ao tratamento de ponteiros e de variáveis compostas: — variáveis estruturadas (vetores e matrizes) e registros —, pois não é possível, em geral, determinar-se estaticamente o elemento particular destas variáveis que está sendo referenciado. Para variáveis estruturadas adotou-se que a definição de um elemento da variável implica a definição da variável; por exemplo, se A é um vetor e ocorre a definição de $A[e]$, onde e é uma expressão, então é considerado que ocorreu a definição de A. Para variáveis do tipo registro adotou-se a mesma abordagem, ou seja, qualquer definição de um campo de um registro é considerado definição do registro. Variáveis do tipo ponteiro são tratadas exatamente como uma variável comum e não é feito o tratamento de definições por *dereferenciação*. Outra vez, não foram tratados esses tipos de definição porque é, em geral, impossível saber estaticamente que objeto de dado [GHE87] estaria sendo definido por dereferenciação. Observe-se que o enfoque adotado para variáveis compostas não é conservador, pois alguns fluxos de dados existentes podem não ser requeridos. Um enfoque conservador, semelhante ao adotado para as variáveis definidas por referência, poderia ser adotado para variáveis compostas. Adicionalmente, pretende-se estender este modelo para tratar definições por deferenciação; uma abordagem usualmente adotada em análise estática [POD90] é considerar um acesso através de um ponteiro como um acesso a todos os objetos que possam ser apontados pelo ponteiro. Pretende-se, adicionalmente, conduzir-se estudos comparativos destas diferentes abordagens.

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

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

Outro fator que pode levar à ocorrência de indefinição de variáveis é a permissão de declaração de variáveis no início de blocos; em geral, as regras de escopo utilizadas determinam que o escopo das variáveis é limitado ao bloco onde as variáveis são declaradas, como nas linguagens C e Algol 68 e, portanto, estariam indefinidas fora do bloco onde são definidas. Este aspecto é muito importante na construção dos grafo(i), pois a seleção de caminhos ou associações deve se restringir ao escopo da variável. Um outro ponto importante é quando ocorre aninhamento de blocos com a ocorrência de declarações de variáveis com o mesmo nome. Considere uma variável v declarada em um bloco m, definida no nó i deste mesmo bloco e redeclarada em um bloco mais interno n. Denotemos por v_m e v_n para diferenciar entre a variável v do bloco m e a do bloco n, respectivamente. Pelas regras de escopo mais usuais, a variável v_m não seria visível no bloco n, ou seja, ela é indefinida

neste bloco e, portanto, jamais seria redefinida ou utilizada até o encerramento da execução dele. Observe-se ainda que qualquer definição de v_n no bloco n seria invisível para v_m. Após a execução do bloco n a variável v_m seria novamente visível e passível de definição e uso. Consequentemente, a ocorrência de declarações de variáveis em blocos mais internos não deve inibir (afetar) a determinação de caminhos ou associações requeridas.

As Figuras 2.8, 2.9 e 2.10 sintetizam os conceitos e hipóteses básicos para a determinação do grafo def a partir do modelo de fluxo de controle dos principais comandos da linguagem L1. Observe-se que para os comandos sequenciais a extensão é óbvia, uma vez que estes comandos estão sempre associados a um único nó. A generalização destas idéias para a inclusão de comandos de desvios condicionais e incondicionais é simples e direta.

2.3.5 Utilização de Arcos Primitivos no Contexto de Fluxo de Dados

Nesta seção, discute-se o uso do conceito de arco primitivo, introduzido por Chusho [CHU87], dentro do contexto de teste estrutural baseado em análise de fluxo de dados.

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

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

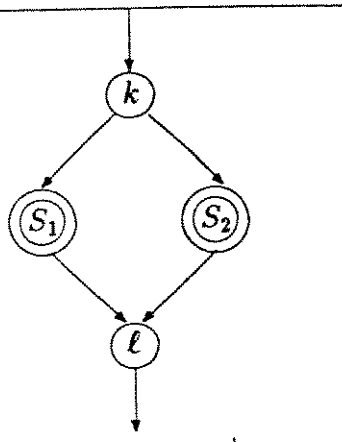
Para ilustrar considere o exemplo de Chusho [CHU87] reproduzido na Figura 2.11. Observe-se que o arco a é herdeiro de b e c, isto é, sempre que b ou $c$forem executados, a também é executado. Note-se também que b é sempre executado quando d ou e são executados; logo, b é herdeiro de d e e e, portanto, não é primitivo.

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

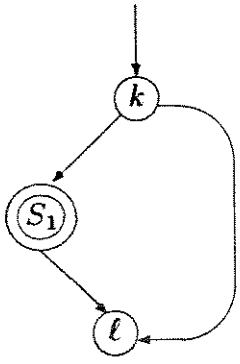
Para poder aplicar o conceito de arcos primitivos em critérios baseados na análise de fluxo de dados foi necessário alterar ligeiramente o algoritmo proposto

Comandos de Seleção

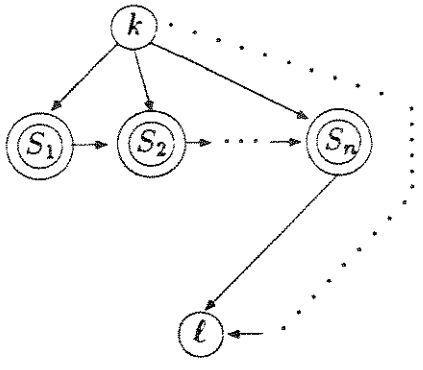
< if > ::= < if-atm >< cond-atm>< statement$_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statement$_2$ >



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



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

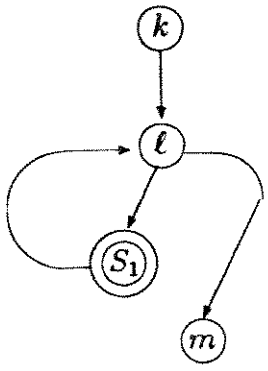


Ao nó *k* são atribuídos o conjunto de variáveis definidas no bloco de comandos associados a este nó e o conjunto de variáveis definidas na condição do case associada a < case-cond-atm >. Ao nó *l* é atribuído o conjunto de variáveis definidas no bloco de comandos associado a a este nó. A extensão dos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

Figura 2.8: Diretrizes para a Expansão do Grafo de Programa para obtenção do Grafodef: Comandos de Seleção.

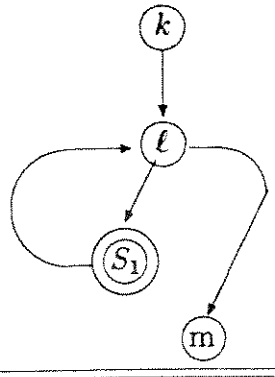
Comandos de Iteração

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



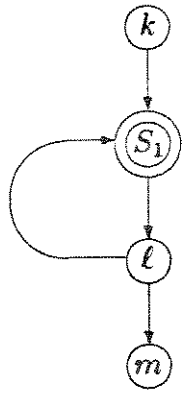
Ao nó k não é associada definição de variável devido ao comando < while >, somente devido aos demais comandos eventualmente associados ao nó k. Ao nó l é associado o conjunto de variáveis definidas na condição associada a < cond-while-atm >. Ao nó m somente é atribuído o conjunto de variáveis definidas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

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



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

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



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

Figura 2.9: Diretrizes para a Expansão do Grafo de Programa para obtenção do Grafodef: Comandos de Iteração.

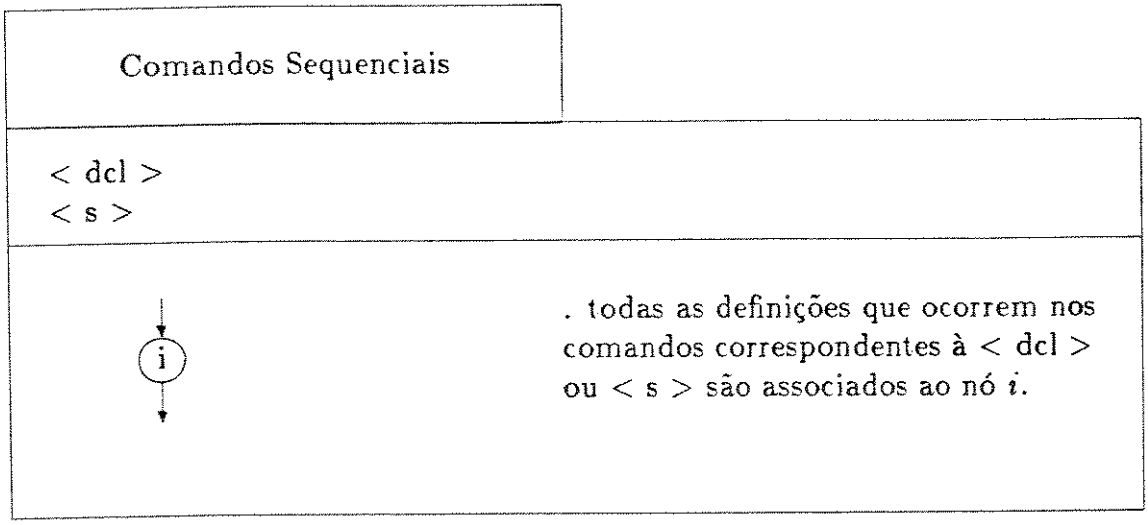


Figura 2.10: Diretrizes para a Expansão do Grafo de Programa para obtenção do Grafodef: Comandos de Sequenciais.

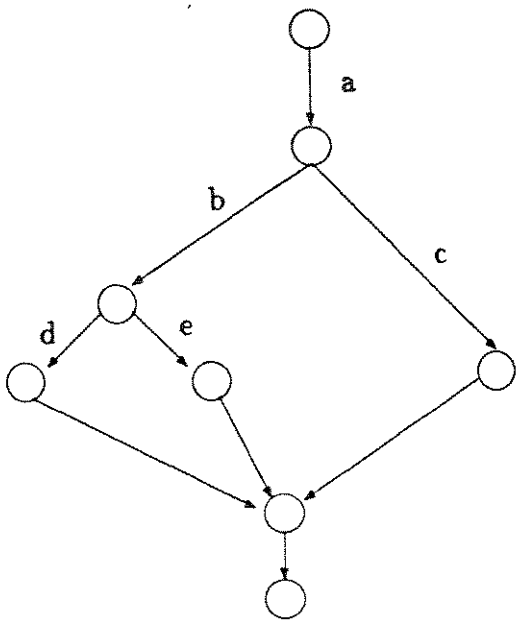


Figura 2.11: GFC com seus arcos primitivos e herdeiros.

por Chusho. Basicamente, essa alteração consiste em não aplicar uma das regras de redução de herdeiros de Chusho.

A seguir, serão apresentados conceitos e definições introduzidos por Chusho [CHU87], necessários para entendimento do algoritmo de Chusho e do algoritmo modificado. A implementação do algoritmo modificado, na POKE-TOOL, é apresentada na Seção 3.6.1.

O Algoritmo de Chusho

Nesta seção serão apresentados, inicialmente, alguns dos conceitos básicos e definições introduzidos por Chusho [CHU87], necessários para entendimento das regras de redução de herdeiros e do algoritmo em si. Posteriormente, as regras e o algoritmo serão detalhados.

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

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

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

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

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

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

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

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

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

Condição 1 Para um GFC *G(N,A,s)*,

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

Regra de Redução R 1 Sob a condição 1, se

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

(x,y) é eliminado de A e x e y são unidos em um único nó, como mostrado na figura 2.12a.

Regra de Redução R 2 Sob a condição 1, se

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

(x,y) é eliminado de A e x e y são unidos em um único nó, como apresentado na Figura 2.12b.

Regra de Redução R 3 Sob a condição 1, se

$$OUT(x) \geq 2$$

e

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

(x,y) é eliminado de A e x e y são unidos em um único nó, como apresentado na Figura 2.12c.

Regra de Redução R 4 Sob a condição 1, se

$$IN(y) \geq 2$$

e

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

(x,y) é eliminado de A e x e y são unidos em um único nó, como apresentado na Figura 2.12d.

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

Algoritmo de Redução de Herdeiros:

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

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

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

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

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

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

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

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

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

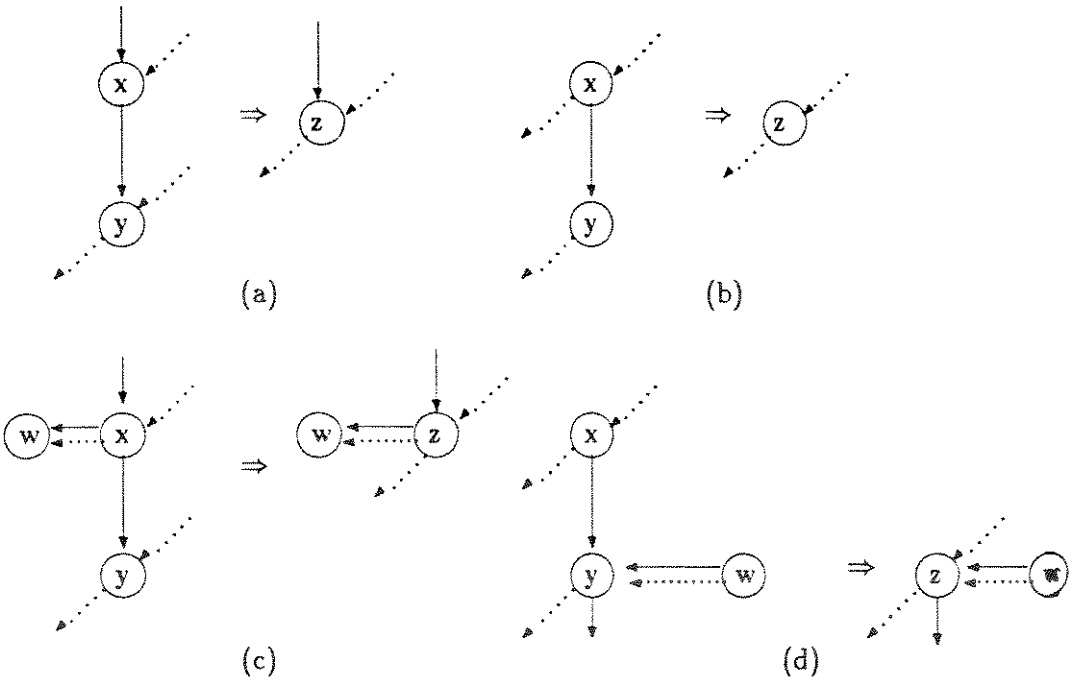


Figura 2.12: Regras para Redução de Herdeiros.

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

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

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

Esse novo algoritmo é denominado *Algoritmo de Redução de Herdeiros de Fluxo de Dados* (REHFLUXDA). O arco ao qual seria aplicada a regra **R3** denomina-se *Arco Primitivo de Fluxo de Dados* (pfd). A aplicação da regra **R3** impossibilitaria a descrição de caminhos livres de laços requeridos pelos critérios baseados em fluxo de dados, em particular pelos critérios todos-potenciais-usos/du e todos-potenciais-du-caminhos. Também não seria possível descrever genericamente uma particular associação de forma que todos os caminhos que a exercitassem fossem completamente identificados. Maldonado [MAL91a, MAL91c] mostra detalhadamente o porquê da eliminação da regra **R3**.

2.3.6 Grafo(i) e Descritores

Com o objetivo de prover-se uma uniformidade à implementação de ferramentas de suporte a testes estruturais, em particular aos critérios Potenciais Usos, foi sugerido o conceito de *grafo(i)* [MAL88b], obtido a partir do grafo def e que fornece "todos" os caminhos livres de definição c.r.a. qualquer variável definida em i para todo nó e todo arco alcançável a partir do nó i. A proposição é de se construir um único *grafo(i)* para cada nó $i \mid defg(i) \neq \phi$, obtendo-se por construção uma minimização de caminhos e associações requeridos. Na realidade, os *grafo(i)* incluem somente os potenciais-du-caminhos a partir do nó i. A Figura 2.13 contém um grafo(i) do exemplo do Apêndice A.

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

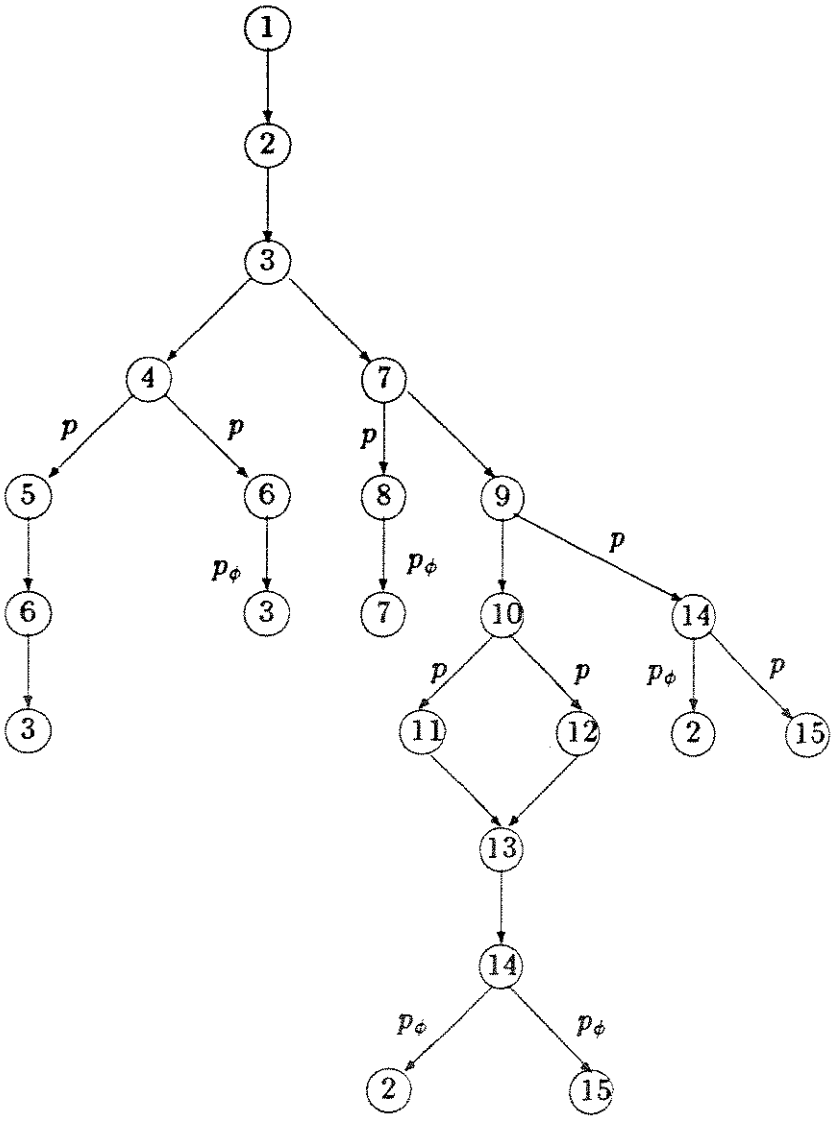


Figura 2.13: Grafo(i) do Exemplo do Apêndice A.

entrada para um módulo avaliador. Este módulo determinaria, em função dos caminhos efetivamente executados, aqueles caminhos e associações requeridos mas não executados.

O algoritmo para a obtenção dos grafo(i) é apresentado na Seção 3.6.3. Esse algoritmo consiste essencialmente em uma busca em profundidade no grafo def a partir de nós que tenham definições de variáveis. Um nó k pertencerá a um grafo(i) se existir pelo menos um caminho livre de definição do nó i para o nó k c.r.a pelo menos uma das variáveis definidas em i. Associa-se a cada nó k do grafo(i) um conjunto de variáveis denominado $deff(k) \subseteq defg(i)$. Para qualquer variável $v \in deff(k)$, qualquer caminho $\pi = (i,\ldots,k)$ no grafo(i), é livre de definição c.r.a. v. Os arcos (k,l) do grafo(i) originados a partir de um arco primitivo do grafo def são caracterizados como arcos primitivos no grafo(i). Um arco (k,l) caracterizado como arco primitivo no grafo(i) não é necessariamente um arco primitivo no grafo(i) se aplicado o algoritmo REXFLUXDA nesse grafo. Ainda, é importante observar que um nó ou um arco (primitivo) do grafo de programa G pode dar origem a mais do que um nó ou um arco (primitivo) no grafo(i); isto decorre do fato que é construído um único grafo(i) para todas as variáveis definidas no nó i [MAL88b]. Então um nó $k \in N$ pode dar origem a q imagens $k_1, k_2, \ldots, k_q$ distintas no grafo(i). Denota-se por $f_{\pi_q}(k)$ a $q-esima$ imagem de k, que ocorre no caminho π_q do grafo(i); se existirem duas ou mais imagens de k em um mesmo caminho π_q, denotam-se por $f_{1\pi_q}(k)$ e por $f_{2\pi_q}(k)$ a primeira e a segunda imagens de k no caminho π_q, respectivamente. É importante salientar que, considerando-se dois caminhos distintos π_1 e π_2, para quaisquer duas imagens distintas no grafo(i), $f_{i\pi_1}(n)$ e $f_{i\pi_2}(n)$, de um nó $n \in N$, $deff(f_{i\pi_1}(n)) \neq deff(f_{i\pi_2}(n))$, $i = 1,2$.

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

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

Seja um grafo $G = (N, A, s)$ e um grafo(i); define-se $N_i = \{n \in N \mid$ existe caminho π livre de definição c.r.a uma variável $v \in defg(i)$ até n, ou seja, existe $f_\pi(n)$ no $grafo(i)\}$; $N_\tau = \{k \in grafo(i) \mid k = f_{2\pi}(n), n \in N$, onde π é um potencial-du-caminho do grafo de programa $G\}$; e $N_\phi = \{k \in grafo(i) \mid deff(k) = \phi\}$. Introduz-se a notação $[i,(j,k),\{v_1,\ldots,v_n\}]$ para representar o conjunto de as-

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

2.3.7 Critério todos-potenciais-du-caminhos

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

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

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

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

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

Se $k_1 = i$, então o descritor é

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

Esses descritores em termos de expressões regulares são, na realidade, correspondentes a autômatos finitos. Os autômatos correspondentes aos descritores dos caminhos requeridos pelo critério todos-potenciais-du-caminhos são ilustrados na Figura 2.14a.

O conjunto N_{if} é igual ao conjunto $N_i/1$, onde $N_i/1$ indica que os elementos de N_{if} podem ocorrer uma única vez; isto quer dizer que este é um conjunto dinâmico, pois, deve ser atualizado sempre que ocorrer um de seus elementos no caminho avaliado. Esta condição deve ser monitorada dinamicamente na implementação dos aceitadores (ver módulo **Avaliador**, Seção 3.7).

2.3.8 Critério todos-potenciais-usos

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

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

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

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

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

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

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

ou, simplesmente, por

$$N^* il.$$

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

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

Esses descritores em termos de expressões regulares são, na realidade, correspondentes a autômatos finitos. Os autômatos correspondentes aos descritores dos

caminhos requeridos pelo critério todo-potenciais-usos são ilustrados na Figura 2.14b.

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

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

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

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

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

ou, simplesmente, por

$$N^*il.$$

O conjunto N_{nv} é redefinido por

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

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

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

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

implica, teoricamente, em também exercitar a associação

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

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

2.3.9 Critério todos-potenciais-usos/du

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

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

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

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

$$N^* il$$

e o conjunto N_{nvlf} por

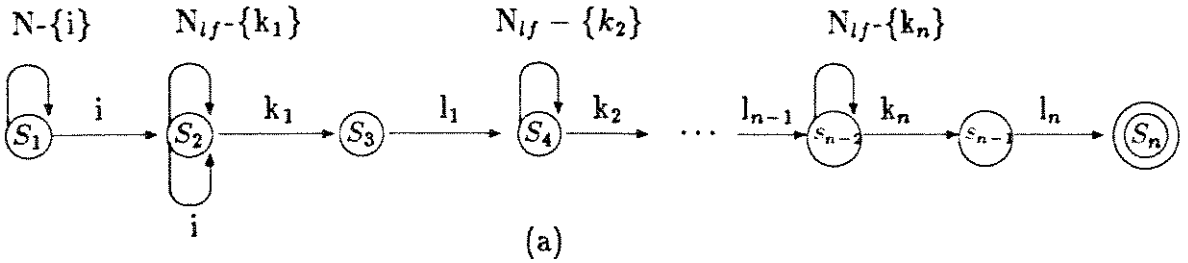
$$N_{nvlf} = N_i - (N_i \cap (\bigcap_{v \in deff(f_{\pi_q}(j))} N_v)),$$

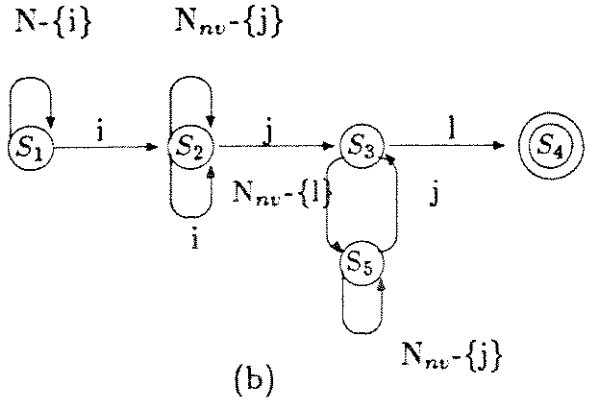
ou seja, o conjunto N_{nvlf} é igual ao conjunto N_{nv}. No entanto, da mesma forma que para o critério todos potenciais-du-caminhos, o conjunto N_{nvlf} é um conjunto dinâmico, isto é, os elementos de N_{nvlf} podem ocorrer uma única vez; ele é atualizado, no processo de avaliação, sempre que ocorrer um elemento pertencente a este conjunto no caminho executado pelo caso de teste. Esta condição deve ser monitorada dinamicamente na implementação dos aceitadores (ver módulo **Avaliador**, Seção 3.7).

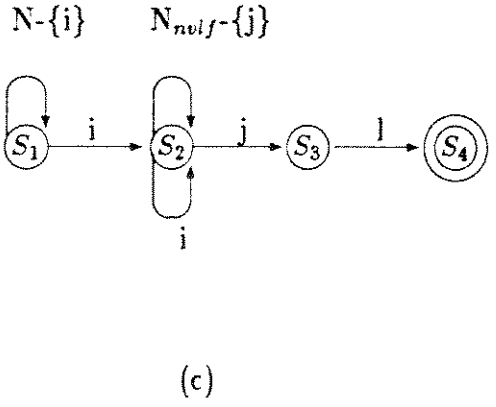
Os autômatos correspondentes aos descritores dos caminhos requeridos pelo critério todos-potenciais-usos são ilustrados na Figura 2.14c.

No contexto da POKE-TOOL duas medidas são fornecidas em decorrência da análise de cobertura. Uma delas fornece um percentual de associações exercitadas em relação às requeridas. Por exemplo, para o critério todos-potenciais-du-caminhos é fornecida uma porcentagem dos caminhos executados com relação aos caminhos requeridos. A outra medida fornece uma média dos percentuais de cobertura dos elementos requeridos em relação a cada grafo(i) [CHA91c, MAL91a].

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







Obs.: Em qualquer estado, a ocorrência do nó i leva o automato pra o estado S_2; da mesma forma, a ocorrência de um nó ($\neq i$) não especificado explicitamente no automato, provoca uma transição para o estado S_1.

Figura 2.14: Automatos Correspondentes aos Descritores dos Critérios Potenciais Usos.

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

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

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

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

Capítulo 3

Projeto/Implementação da POKE-TOOL

A POKE-TOOL é uma ferramenta que apóia a aplicação dos critérios Potenciais Usos para o teste de unidades. As unidades são entendidas aqui como procedimentos de uma linguagem procedural.

A ferramenta pode ser configurada para testar programas escritos em linguagens procedurais com gramáticas livre de contexto que satisfaçam algumas limitações. Basicamente, para configurar a ferramenta, é necessário que o configurador especifique o analisador léxico e o analisador sintático para a linguagem alvo. Em [CHA91a] é descrito como realizar estas tarefas; uma síntese desse processo é fornecida no Capítulo 4.

A POKE-TOOL é uma ferramenta interativa cuja operação é orientada para *sessão de trabalho*. Na sessão de trabalho, o usuário realiza todas as tarefas de teste, a saber: análise estática da unidade; preparação para o teste; submissão de casos de teste; avaliação de casos de teste; e gerenciamento dos resultados de teste. Na POKE-TOOL, a sessão de trabalho é dividida em duas fases: uma *estática* e outra *dinâmica*. Na fase estática a ferramenta analisa o código fonte, obtém as informações necessárias para a aplicação dos critérios e instrumenta o código fonte, com inserção de pontas de prova (instruções de escrita), que produzem um "trace" do caminho executado, gerando uma nova versão da unidade em teste — *versão instrumentada* —, que viabiliza a posterior avaliação da adequação de um dado conjunto de casos de teste. Terminada essa fase, a POKE-TOOL apresenta, sob solicitação do usuário, entre outras coisas, o conjunto de caminhos requeridos pelo critério *todos potenciais-du-caminhos* e o conjunto de associações requeridas pelo critério *todos potenciais-usos* e *todos potenciais-usos/du*. Com estas informações o usuário pode projetar seus casos de teste a fim de que eles executem os caminhos ou associações exigidos.

A fase dinâmica consiste no processo de execução e avaliação de casos de teste.

Porém, antes de executar os casos de teste, é necessário que se gere o programa executável a partir da versão instrumentada da unidade a ser testada. A POKE-TOOL, depois da execução dos casos de teste, realiza a avaliação destes de acordo com um dos três critérios Potenciais Usos. O resultado da avaliação é um conjunto de caminhos ou associações que restam ser executados para satisfazer os critérios e o percentual da cobertura do conjunto de casos de teste. Ainda nesta fase, a ferramenta fornece o conjunto de caminhos ou associações que foram executados, as entradas e saídas, bem como os caminhos percorridos por cada um dos casos de teste. O processo de execução/avaliação deve continuar até que todos os caminhos ou associações restantes tenham sido satisfeitos (executados ou detectada a sua não executabilidade). Eventualmente, o usuário pode querer interromper a sessão de trabalho sem ter atingido uma das duas situações anteriores. Neste caso, a POKE-TOOL provê meios para a interrupção da sessão, armazenamento dos dados gerados e do estado da ferramenta até o momento. Posteriormente, pode-se recuperar a sessão de trabalho e recomeçar os testes.

A POKE-TOOL é composta de vários módulos que se comunicam através de *arquivos*. Esses módulos implementam *funções* ou parte de funções descritas anteriormente na arquitetura da ferramenta. Na Figura 3.1 é apresentado um diagrama contendo os módulos e os diversos produtos gerados. Nessa figura, os *retângulos* representam os módulos, os *losangos* representam as entradas fornecidas pelos usuários à ferramenta e os *círculos* os produtos gerados; as linhas tracejadas representam o *fluxo de controle* e as linhas cheias o *fluxo de informação*.

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

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

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

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

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

O módulo **executa caso de teste**, como o próprio nome diz, controla a execução dos casos de teste salvando as entradas, a saída e o caminho execu-

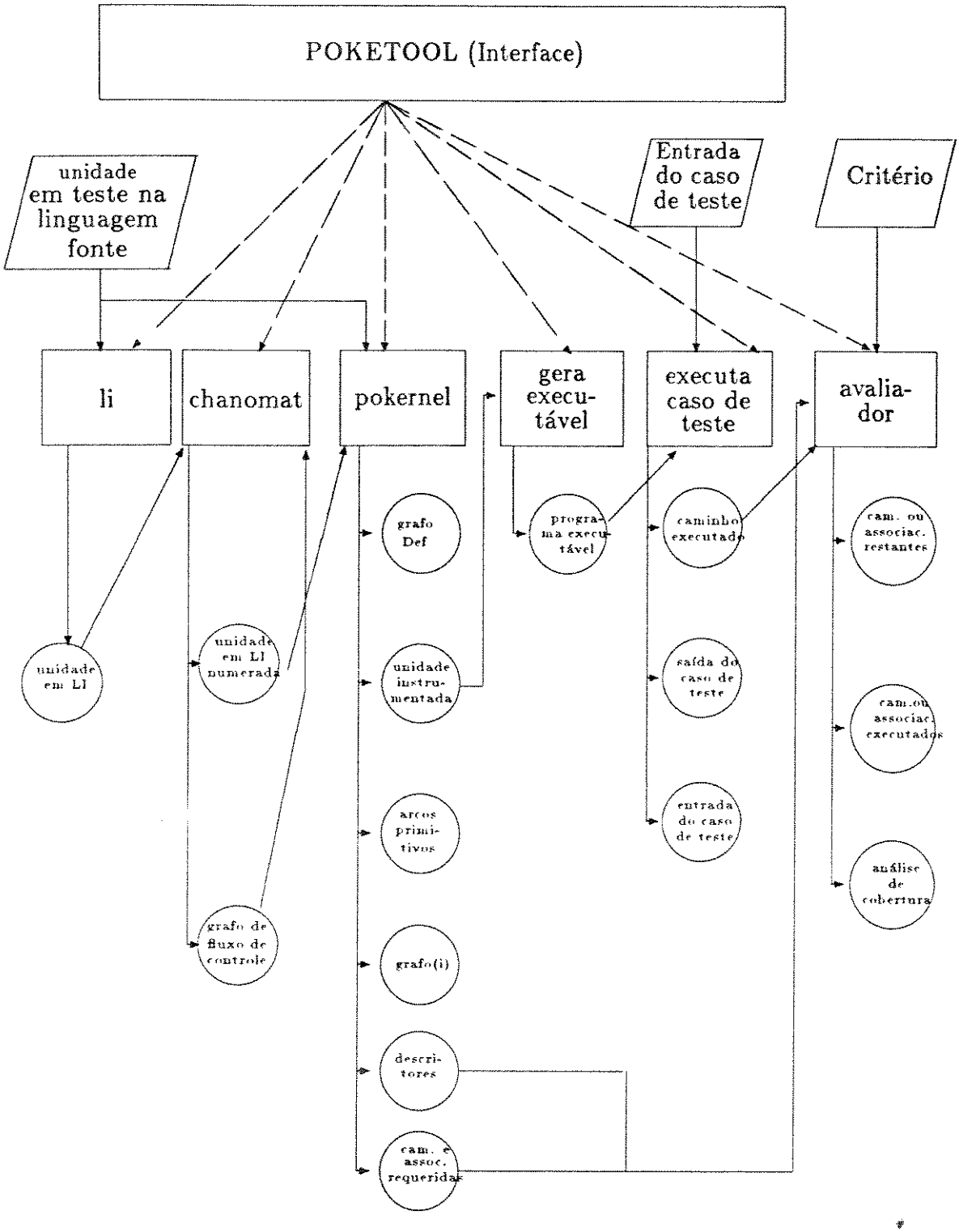


Figura 3.1: Projeto da POKE-TOOL

tado para cada caso de teste; implementa a função *Execução do Programa* da POKE-TOOL.

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

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

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

3.1 Módulo Poketool

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

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

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

A interface é dividida em telas que representam as diversas atividades de teste. A Figura 3.2 apresenta uma hierarquia das telas da POKE-TOOL.

É possível observar, nessa figura, que a própria hierarquia de telas da interface possui uma semelhança muito grande com os módulos constituintes da ferramenta e, por si só, já estabelece a sequenciação das atividades de teste.

A tela INICIAÇÃO prepara o ambiente para o início do processo de teste. Por exemplo, é nessa tela que é perguntado ao usuário se ele deseja iniciar uma nova sessão de trabalho ou recomeçar uma anterior; é também na INICIAÇÃO que é recebido o nome da unidade a ser testada e são ativados os módulos **li**, **chanomat** e **pokernel**[1] para a análise estática da unidade (caso o usuário deseje iniciar uma nova sessão de trabalho).

A tela MENU PRINCIPAL faz a sequenciação das atividades posteriores à análise estática da unidade, ativando as demais telas desde que algumas restrições tenham sido observadas. Assim, o usuário não conseguirá ativar as telas EXECUTA CASO DE TESTE e AVALIAÇÃO se não tiver sido gerada a unidade

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

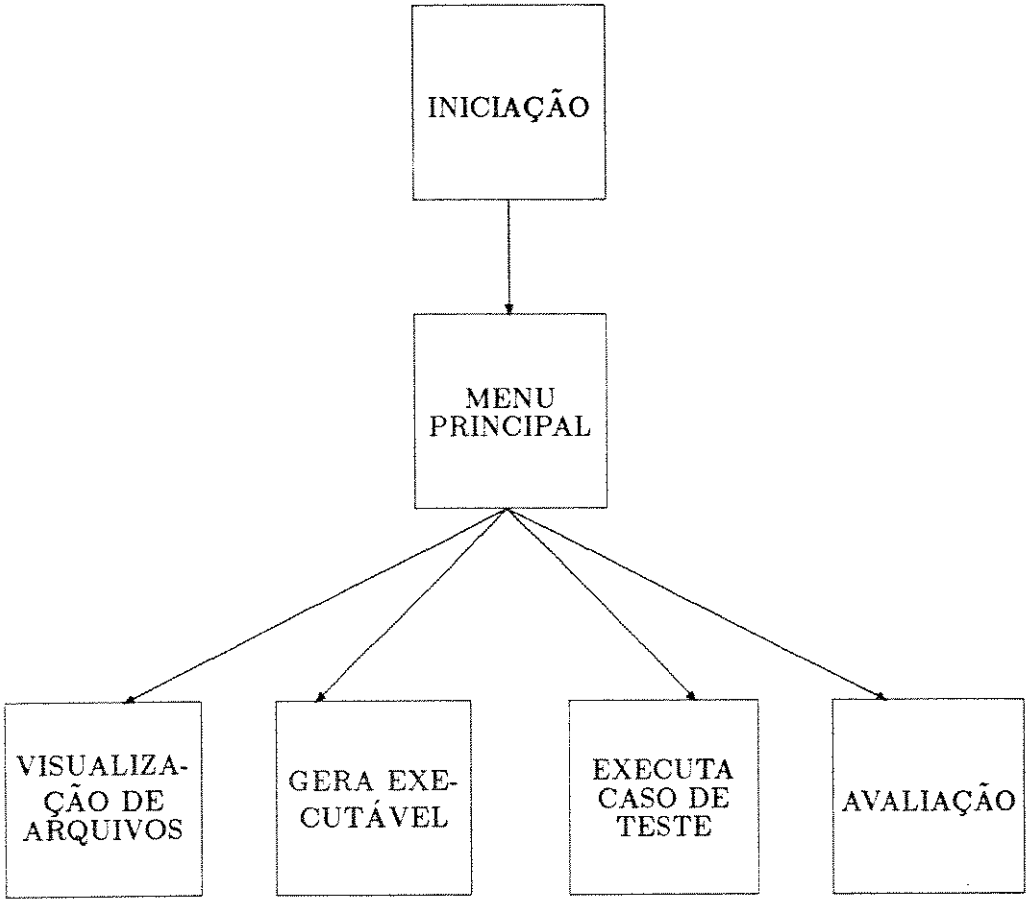


Figura 3.2: Hierarquia das telas da POKE-TOOL

modificada para teste. A tela MENU PRINCIPAL contém, então, 5 opções; quatro delas causam a seleção das atividades controladas pela POKE-TOOL com a ativação da tela correspondente. A outra - *termina sessão* - termina a sessão de trabalho salvando opcionalmente, em um diretório, os resultados e informações da sessão de teste.

Uma dessas atividades é a VISUALIZAÇÃO DE ARQUIVOS gerados pela ferramenta e desempenhada através da tela de mesmo nome. Essa tela apresenta arquivos gerados tanto na fase estática quanto na fase dinâmica[2] . Tipicamente, os resultados da fase estática apresentados são: o conjunto de arcos primitivos, o grafo def, a unidade modificada para teste, o conjunto de caminhos requeridos pelo critério todos-potenciais-du-caminhos e as associações requeridas pelos critérios todos-potenciais-usos e todos-potenciais-usos/du. Da fase dinâmica são apresentados: as entradas de parâmetros para cada caso de teste (parâmetros aqui se referem aos parâmetros passados na linha de comando para o sistema operacional quando o programa executável, que contém a unidade em teste, é executado), entradas efetuadas via teclado (caso isto ocorra na unidade em teste), as saídas na tela, os caminhos executados, o conjunto de potenciais du-caminhos efetivamente executados, o conjunto de associações efetivamente executadas para o critério todos-potenciais-usos e para o critério todos-potenciais-usos/du. Note-se, entretanto, que algumas restrições devem ser observadas antes de se apresentar alguns desses arquivos. Assim, não são apresentadas as entradas, as saídas ou caminhos executados dos casos de teste se nenhum deles foi executado; não é possível apresentar os conjuntos de caminhos e associações executados se não houve a avaliação de pelo menos um caso de teste. A consistência entre as opções de visualização e o estado atual do processo de teste é verificada pela POKE-TOOL.

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

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

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

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

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

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

(1) nome da unidade em teste;

(2) quantos casos de teste foram executados;

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

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

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

(6) se a unidade aceita parâmetros e

(7) se ela aceita entrada pelo teclado.

No Capítulo 4 é apresentado um exemplo de utilização da POKE-TOOL com a descrição em "hard copy" da interface da ferramenta.

3.2 Módulo Gera Executável

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

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

3.3 Módulo Executa Caso de Teste

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

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

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

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

3.4 Módulo Li

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

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

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

O módulo li é um programa executável que pode ser configurado para fazer a tradução de uma determinada linguagem para a LI. A configuração do módulo li deve ser feita por um usuário configurador e envolve a criação de tabelas e rotinas específicas para a tradução da linguagem em questão para a LI. Carnassale [CAR91] descreve como criar essas tabelas e rotinas. Na Seção 2.3.1 e no Apêndice A existem exemplos de unidades traduzidas para a LI pelo módulo li da POKE-TOOL.

3.5 Módulo Chanomat

Este módulo calcula o grafo de fluxo de controle (GFC), segundo o modelo proposto na seção 2.3.2, e também produz uma versão da unidade em LI com um campo a mais nos átomos indicando a qual nó pertence o átomo.

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

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

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

A seguir, é apresentado o arquivo que contém o GFC para o exemplo da Seção 2.3.1, bem como a nova versão da unidade em LI. Se esse exemplo estiver contido no arquivo KER.C, o GFC estará descrito no arquivo KER.GFC, a versão em LI no arquivo KER.LI e a nova versão em LI no arquivo KER.NLI.

KER.GFC

```
24

1
   2 0
2
   3 4 0
3
   2 0
4
   5 21 0
5
   6 7 8 9 10 11 12 13 14 15 16 17 18 19 0
6
   7 0
7
   8 0
8
   9 0
9
   10 0
10
   11 0
11
   12 0
12
   13 0
13
   14 0
14
   15 0
15
   20 0
16
   17 0
17
   18 0
18
   20 0
19
   20 0
20
```

```
   4 0
21
   22 0
22
   23 24 0
23
   22 0
24
   0
```

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

A nova versão da LI para o exemplo é dada abaixo.

KER.NLI

```
$DCL                1      1     11      1
{                   1     54      1      2
$DCL                1     58     37      3
$SO1                1     98     19      4
$FOR                1    120      3      5
$SO2                1    124      4      5
$C(O1)O1            2    129      6      5
$SO3                3    136      3      5
{                   3      0      0      0
$SO4                3    145     14      6
}                   3      0      0      0
$WHILE              4    164      5      8
$C(O1)O2            4    169     24      8
{                   5      0      0      0
$CASE               5    197      6      9
$CC                 5    203      3      9
{                   5    207      1      9
$ROTC               6    212      9     10
{                   6      0      0      0
}                   6      0      0      0
$ROTC               7    225      9     11
```

{	7	0	0	0
}	7	0	0	0
$ROTC	8	238	9	12
{	8	0	0	0
}	8	0	0	0
$ROTC	9	251	9	13
{	9	0	0	0
}	9	0	0	0
$ROTC	10	264	9	14
{	10	0	0	0
}	10	0	0	0
$ROTC	11	277	9	15
{	11	0	0	0
}	11	0	0	0
$ROTC	12	290	9	16
{	12	0	0	0
}	12	0	0	0
$ROTC	13	303	9	17
{	13	0	0	0
}	13	0	0	0
$ROTC	14	316	9	18
{	14	0	0	0
}	14	0	0	0
$ROTC	15	329	9	19
{	15	0	0	0
$SO5	15	340	16	19
$BREAK	15	371	6	20
}	15	0	0	0
$ROTC	16	380	9	21
{	16	0	0	0
}	16	0	0	0
$ROTC	17	392	10	22
{	17	0	0	0
}	17	0	0	0
$ROTC	18	405	10	23
{	18	0	0	0
$SO6	18	420	9	24
$BREAK	18	434	6	25
}	18	0	0	0
$ROTD	19	443	8	26
{	19	0	0	0

$SO7	19	456	9	27
$BREAK	19	470	6	28
}	19	0	0	0
}	20	478	1	29
}	20	0	0	0
$SO8	21	481	19	30
$FOR	21	502	3	31
$SO9	21	506	4	31
$C(O1)O3	22	511	6	31
$S10	23	518	3	31
{	23	0	0	0
$S11	23	526	25	32
}	23	0	0	0
$S12	24	553	59	33
}	24	614	0	34

3.6 Módulo Pokernel

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

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

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

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

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

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

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

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

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

O primeiro passo desse módulo é calcular os arcos primitivos do GFC. O conceito de arco primitivo foi introduzido por Chusho [CHU87] e visa determinar um conjunto de arcos do GFC, cuja execução implica na execução de todos os arcos do GFC. No contexto da POKE-TOOL, os arcos serão utilizados para descrever os caminhos e associações exigidos pelos critérios Potenciais Usos (PU). Com a utilização de arcos primitivos pretende-se otimizar o processo de avaliação da cobertura dos casos de teste e prover um mecanismo uniforme para a avaliação da adequação de conjuntos de casos de teste em relação a critérios de teste estruturais. O algoritmo utilizado para calcular esses arcos — Algoritmo de Redução de Herdeiros de Fluxo de Dados — está baseado no algoritmo proposto por Chusho [CHU87] ligeiramente modificado para se adequar aos requisitos dos critérios baseados em fluxo de dados, em particular, dos critérios PU (ver Seção 2.3.5).

Outra atividade é determinar o *grafo def*; para determiná-lo é necessário fazer análises sintática e semântica dos comandos que compõem cada nó do GFC para identificar quais variáveis estão sendo definidas nestes nós. O *grafo def* consiste na extensão do grafo de programa, realizada de acordo com os conceitos e diretrizes discutidos na Seção 2.3.4. Durante a determinação do grafo def é produzida a unidade modificada para teste onde estão inseridas as pontas de prova que registram o caminho percorrido pelo caso de teste. Com o grafo def é possível determinar os *grafo(i)* para cada nó do GFC que possui definição de variáveis. A partir dos grafo(i) e dos arcos primitivos, são obtidos os caminhos e as associações requeridos pelos critérios PU, bem como os descritores utilizados na avaliação.

O módulo **pokernel** é um programa executável que recebe os arquivos mencionados acima como entrada. Entretanto, o **pokernel** é também dividido em sub-módulos que realizam as diversas atividades descritas acima. O grafo de chamada apresentado na Figura 3.3 descreve os sub-módulos do módulo **pokernel** e os arquivos que cada um gera.

O sub-módulo **cal_prim** calcula os arcos primitivos utilizando o algoritmo de *redução de herdeiros de fluxo de dados*, que será apresentado a seguir. O sub-módulo **parserli** tem a função de determinar o grafo def e a unidade em teste modificada. Finalmente, o módulo **des_pot_du_cam** determina os grafo(i) e, através destes, os caminhos e associações requeridos e os descritores utilizados para avaliar a adequação de conjuntos de casos de teste em relação aos critérios PU.

A seguir, será apresentada a descrição mais detalhada dos sub-módulos do módulo **pokernel**.

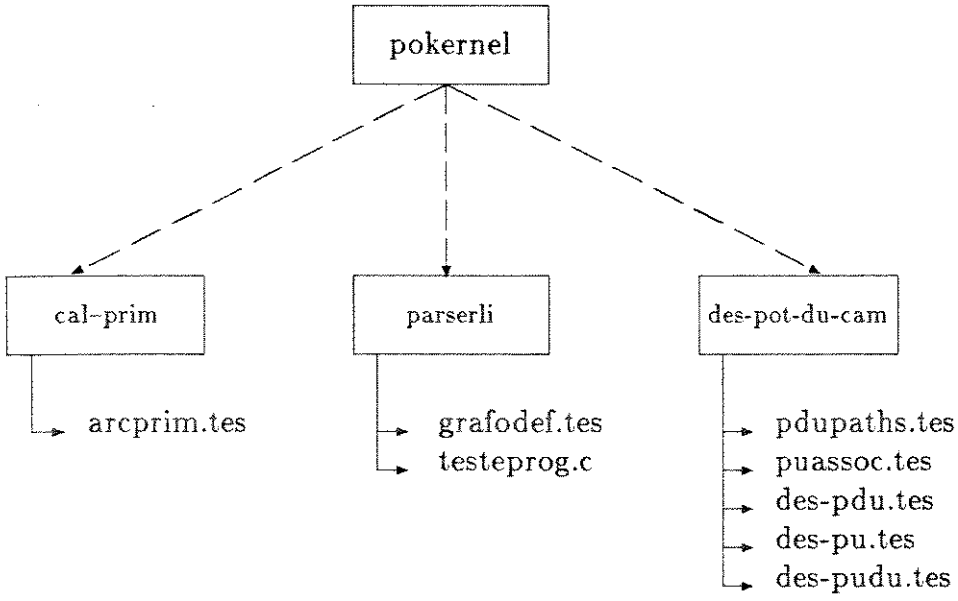


Figura 3.3: Grafo de Chamada do Módulo **pokernel**.

3.6.1 Cálculo dos Arcos Primitivos (sub-módulo cal_prim)

Este sub-módulo faz o *cálculo dos arcos primitivos* do GFC da unidade, de acordo com o algoritmo de redução de herdeiros de fluxo de dados.

Algoritmo de Redução de Herdeiros de Fluxo Dados:

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

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

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

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

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

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

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

8) O passo 7) é repetido até nenhum arco com marca de herdeiro seja encontrado.

Implementação do Algoritmo REHFLUXDA na POKE-TOOL

Este algoritmo recebe como entrada um GFC e retorna como saída o conjunto de arcos primitivos de fluxo de dados desse GFC.

Para implementar um GFC na POKE-TOOL foi utilizado a representação em forma de listas de sucessores, ou seja, os grafos são vetores onde os elementos desses vetores são registros que contêm o número do nó do GFC e um apontador para a lista de nós sucessores desse nó. Os elementos da lista de sucessores são compostos por três campos. O primeiro campo indica o número do nó sucessor; o segundo indica se o arco constituído pelo nó fonte (designado pelo elemento do vetor) e o nó sucessor é um arco primitivo, herdeiro ou se o arco possui uma marca de herdeiro; e o terceiro campo é um apontador para o próximo elemento da lista.

Para aplicar a regra **R1**, basta percorrer todos os arcos do GFC e verificar as condições de **R1**. Os valores *IN(x)* e *OUT(x)* são muito simples de calcular.

No caso de *IN(x)*, é construído o mesmo GFC só que na forma de uma lista de predecessores e para determinar o *IN(x)* de um nó x basta calcular o número de elementos da lista de predecessores. Raciocínio idêntico é feito para *OUT(x)*, utilizando o GFC na forma de lista de sucessores. Em seguida, o algoritmo percorre os arcos do GFC "reduzindo" os arcos que satisfazem a regra **R1**. O verbo reduzir está entre aspas porque nessa implementação do algoritmo não ocorre a redução do grafo como proposto por Chusho. O que é feito é que o arco que satisfaz a regra **R1** simplesmente tem seu estado mudado de *primitivo* para *herdeiro* pois, antes de aplicar qualquer regra, todos os arcos são primitivos.

A aplicação da regra **R2** não é tão simples como a aplicação de **R1**. O principal problema é determinar *IN(y)* e *OUT(y)*, mas para o grafo "reduzido". Novamente, o que se faz é percorrer todos os arcos primitivos do GFC e aplicar **R2**. Na hora de determinar *IN(y)* (*OUT(y)*) é importante lembrar que o número de elementos das listas de predecessores (sucessores) não representam mais o número de arcos entrando (saindo) de um nó x porque os arcos herdeiros que entram (saem) do nó x não existem mais; ou seja, para efeito de cálculo de *IN(y)* e *OUT(y)*, os nós fonte e sucessor de um arco herdeiro são um único nó.

Para solucionar esse problema foi adotada como solução fazer uma pesquisa recursiva. Essa pesquisa funciona da seguinte maneira para o cálculo de *IN(y)*. Inicialmente, toma-se a representação do GFC na forma de lista de predecessores e verifica-se quantos arcos primitivos entram no nó x; os arcos primitivos são arcos que efetivamente entram em x, portanto, o valor de *IN(y)* é incrementado quando se encontra um arco primitivo. Quando é encontrado um arco herdeiro, é feita a mesma pesquisa só que para o nó fonte do arco herdeiro, pois esse nó está unido a x e, assim por diante, até encerrar os arcos que entram em x. Adicionalmente, é necessário que seja feita uma pesquisa semelhante à descrita anteriormente para os nós alvo dos arcos herdeiros que saem de x, pois esses nós também estão reduzidos a x. Considerando o esboço de GFC da Figura 3.4 e imaginando que se queira saber o número de arcos que entram no nó 4, vê-se que não é suficiente olhar somente o número de arcos que entram em 4; é necessário verificar-se o número de arcos primitivos que entram em 4, 2, 3 e 5 pois, na verdade, esses nós estão reduzidos a um único nó.

Para determinar *OUT(y)* a pesquisa é análoga. Feita a determinação de *IN(y)* e *OUT(y)*, a aplicação de **R2** é idêntica à de **R1**.

A regra **R4** é a mais sofisticada e é a que exige mais em termos de processamento. Para aplicar essa regra, é necessário que seja calculado o conjunto de dominadores *DOM(x)* para todo nó x do GFC. Para calcular esses conjuntos foi utilizado um algoritmo genérico (para qualquer GFC) contido em [HEC77]. Para calcular o conjunto *IN(y)* é feita a mesma pesquisa realizada para **R2**, porém, com uma ligeira diferença. No caso de **R4**, é necessário saber o número de arcos que entram em y bem como os nós fonte que constituem esses arcos. Para resolver

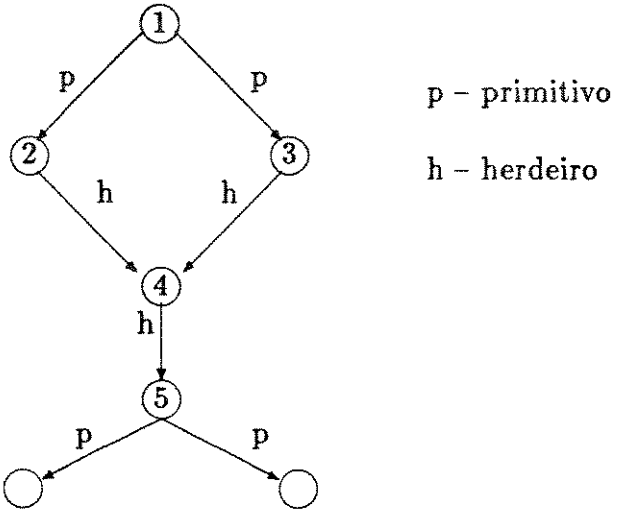


Figura 3.4: Esboço de GFC e seus arcos primitivos e herdeiros.

esse problema, basta acrescentar em uma lista todo nó fonte de arco primitivo que entra em y (ou em algum nó reduzido a y) quando é feita a pesquisa para determinar *IN(y)*.

Com o valor de *IN(y)*, a lista de nós que efetivamente entram em y e o conjunto de dominadores dos nós do GFC é possível verificar se o arco (x,y) é candidato a receber um marca de herdeiro, ou seja, satisfaz **R4**; para receber a marca de herdeiro, o arco (x,y) deve ainda satisfazer pelo menos uma das seguintes condições:

(a) existir pelo menos um arco sem marca de herdeiro entre os arcos que saem de y; ou

(b) existir pelo menos um caminho no sentido contrário dos arcos do GFC que ligue inversamente um dos arcos entrada de y até y, e que inclua pelo menos um arco sem marca de herdeiro.

O arco candidato (x,y) não é elegível para participar de um caminho da condição (b).

Antes de mostrar as estratégias para verificar essas duas condições deve-se definir o que seja um arco com marca de herdeiro. Um arco, na implementação do algoritmo REHFLUXDA da POKE-TOOL, pode assumir três estados: *primitivo, herdeiro e "com marca de herdeiro"*. Inicialmente, todos os arcos são primitivos; quando um arco satisfaz a regra **R1** ou a regra **R2**, ele passa a ter o estado de herdeiro; ou seja, não "existe" mais, o grafo foi "reduzido". Um arco pode

possuir um terceiro estado que é o estado "de possuir uma marca de herdeiro", isto é, este arco satisfaz as condições de **R4** e uma das condições (a) ou (b). Para verificar a condição (a), deve-se fazer uma pesquisa recursiva no nó *y*, e nos demais nós reduzidos a ele. Essa pesquisa é análoga ao cálculo de *IN(y)*. Para verificar a condição (b), é feita uma busca em profundidade inversa[3] tendo o nó *y* como nó partida e chegada. Durante essa busca são mantidas duas variáveis: uma delas indica se no caminho existe algum arco sem marca de herdeiro e a outra indica o sucesso da busca; a variável que indica o sucesso é feita verdadeira se foi encontrado um caminho de *y* a *y* e a variável que indica ausência de marca de herdeiro é verdadeira para este caminho.

Depois de verificada a aplicação de **R4**, percorrem-se os arcos do GFC mudando o estado dos arcos que possuem marca de herdeiro para herdeiro. Os arcos que restaram com o estado *primitivo* são os arcos primitivos do GFC. Lembre-se que estes arcos primitivos incluem aqueles arcos que seriam eventualmente eliminados pela aplicação da regra **R3**, ou seja, os arcos primitivos de fluxo de dados.

3.6.2 Determinação do Grafo Def e a da Unidade Instrumentada (sub-módulo parserli)

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

A determinação do grafo def utilizando a LI é simples porque cada declaração, comando sequencial, ou expressão condicional, é representado por um átomo da LI, onde esse átomo possui ponteiros para o código fonte e um número que indica o seu nó no GFC. Dessa maneira, basta tomar os ponteiros para o código fonte e analisar esse trecho da unidade, verificando a presença de definições e associando as definições, se existerem, ao nó.

O processo de inserir pontas de prova na unidade em teste, gerando a unidade modificada para teste, é chamado *Instrumentação*. Esse processo depende essencialmente da identificação dos comandos que alteram o fluxo de controle do programa em LI porque o processo constitui-se na inserção de pontas de prova e alguns comandos e não da análise dos átomos. Portanto, a Instrumentação vai exigir somente a análise sintática da unidade em LI.

Essas duas tarefas são realizadas pelo sub-módulo **parserli** concomitantemente visto que esse módulo é basicamente um analisador sintático descendente da LI. O analisador da LI vai reconhecendo os átomos e comandos da LI e tomando as

[3] O termo inversa indica que o GFC será percorrido no sentido contrário dos arcos.

ações necessárias para a realização das tarefas. Por exemplo, quando o analisador da LI identifica um comando sequencial *$S*, ele chama um analisador sintático e semântico específico para comandos sequenciais da linguagem fonte da unidade em teste. O mesmo ocorre quando é identificada uma declaração de variável (*$DCL*) ou uma condição (*$C*) da LI. Quando o analisador da LI identifica um comando "while" ele sabe que deve inserir uma ponta de prova dentro do corpo do "while" para indicar a passagem pelo nó da condição, e uma ponta de prova, correspondente a esse mesmo nó, imediatamente depois do corpo do "while" para indicar a passagem pelo nó da condição quando da saída do "loop". Como se vê, a base dos dois processos é o analisador sintático da LI porque é ele quem vai chamar o analisador específico e indicar onde colocar as pontas de prova. Por isso que o nome desse sub-módulo é **parserli**. A extensão do GFC e a Instrumentação são detalhados abaixo.

Extensão do Grafo de Fluxo de Controle

O objetivo da extensão do GFC é gerar o *grafo def*. Para gerar esse grafo, como já foi dito, é preciso fazer uma análise sintática e semântica específica para o código fonte da unidade. O analisador sintático e semântico da POKE-TOOL é específico para a linguagem fonte da unidade em teste; a tarefa básica desse analisador é identificar as variáveis que são definidas nos comandos associados a um dado nó. Essa tarefa, aparentemente simples, envolve a identificação das construções que provocam uma definição de variável na linguagem da unidade em teste e as variáveis, propriamente ditas, envolvidas nessas construções. Portanto, essa tarefa requer um conhecimento da semântica da linguagem das unidades em teste.

A POKE-TOOL provê um analisador léxico e um analisador sintático genéricos que devem ser configurados para realizar essa tarefa. O analisador léxico é baseado na utilização de autômatos finitos [SET81] e o analisador sintático baseia-se na análise sintática descendente genérica através de grafos sintáticos [WIR76]. Esses dois analisadores identificam as variáveis sendo definidas e as associam ao nó correspondente. Para realizar essa tarefa, os analisadores possuem rotinas que manipulam as estruturas de dados que representam o *grafo def*. Como configurar os analisadores léxico e sintático é descrito em [CHA91a].

A versão atual da POKE-TOOL foi desenvolvida para a linguagem C. A configuração para a linguagem C envolveu determinar o modelo de dados (conjunto de construções que causam definição de variáveis) de C segundo as diretrizes estabelecidas na Seção 2.3.4, o tratamento da estrutura de blocos dessa linguagem e a configuração dos analisadores léxico e sintático. O tratamento da estrutura de blocos em C é compatível com o da maioria das linguagens do estilo ALGOL.

A seguir, serão apresentados o modelo de dados estabelecido para C e as es-

truturas de dados utilizadas para implementação do grafo def e o tratamento da estrutura de blocos de C. A configuração dos analisadores de C é apresentada como exemplo em [CHA91a].

Modelo de Fluxo de Dados de C

Para aplicação dos Critérios Potenciais Usos em programas escritos em C, ou seja, configurar a ferramenta POKE-TOOL para a linguagem de programação C, devem ser considerados praticamente todos os pontos discutidos acima quanto ao modelo de dados, pois esta linguagem possui todas as características abordadas, a menos da passagem de parametros por nome. A linguagem C permite o aninhamento de blocos, a definição de variáveis no início de blocos, suporta o uso de todos os tipos de variáveis simples e compostas e de ponteiros. A seguir são discutidas as características mais específicas desta linguagem.

Na linguagem C, uma variável presente no lado esquerdo de um dos diversos comandos de atribuição é assumido como uma *definição* . Adicionalmente, uma variável seguida ou precedida de um operador de *incremento* ou *decremento* ("++" e "--") também o é.

Em C, comandos de entrada são realizados através de chamadas a procedimentos padrões (Ex: scanf, getchar, etc), e portanto as definições das variáveis que ocorrem nas chamadas destes procedimentos são entendidas como definidas por referência.

Com relação às variáveis compostas: estruturadas (vetores e matrizes) e registros foi assumido que uma definição de um elemento da variável implica na definição da variável. Variáveis ponteiros são tratadas exatamente como uma variável comum e não é feito o tratamento de definições por *dereferenciação* .

Para as chamadas de função o modelo tem algumas peculiaridades. A linguagem C considera que todos os parâmetros são passados por valor. Entretanto, quando é desejado forçar uma passagem por referência, o que se faz é passar por valor o endereço da variável. Para obter o valor do endereço de uma variável, ela deve ser precedida pelo operador "&". Variáveis estruturadas como vetores e matrizes têm a caracterítica de ter o valor do endereço passado quando o nome da variável é um parâmetro da chamada de função. Assim, foi assumido que uma variável passada como parâmetro, a princípio, se trata apenas de um uso e, portanto, sem interesse para os critério Potenciais Usos. Porém, quando a variável passada como parâmetro é precedida do operador "&" ou é o nome de uma variavel estruturada, é assumido que a variável em questão poderia ter sido definida por referência.

A Tabela 3.1 resume o modelo de dados assumido para a linguagem C.

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

Tabela 3.1: Modelo de Dados da Linguagem C para obtenção do Grafo Def.

Implementação do Grafo Def

O grafo def é mantido através da estrutura de dados responsável pelo GFC e por uma estrutura de dados muito simples chamada **info_no**. A **info_no** é um vetor cujos elementos contêm várias informações sobre cada nó do GFC. Para acessar as informações do nó *i* do GFC, deve-se consultar o *i-ésimo* elemento de **info_no**.

Entre as informações dos elementos de **info_no** encontram-se três conjuntos. O conjunto de variáveis definidas (**defg**), o conjunto de variáveis possilvemente definidas por referência (**def_ref**) e o conjunto de variáveis indefinidas (**undef**) no nó. Esses conjuntos são representados atráves de "bits" (conjunto de "bits"), onde cada "bit" corresponde a uma variável.

Inicialmente, esses conjuntos de "bits" são ajustados com o valor "0". Cada variável declarada na unidade em teste é inserida em uma *tabela de símbolos* (**tab_var_def** [CHA91a]) onde a variável recebe um *número de identificação* e são armazenadas sua *descrição* (cadeia de caracteres da variável), sua *estrutura* (e.g.: simples, vetor ou ponteiro) e seu *nível de aninhamento*. O número de identificação da variável serve para identificar o "bit" associado à variável nos conjuntos **defg**, **def_ref** e **undef**. Quando é analisado um *$S* ou *$C* e é encontrada uma das situações descritas acima no *Modelo de Dados*, ocorre a inserção da variável em questão no conjunto **defg** ou **def_ref** do nó associado a *$S* ou *$C*. Para saber qual o número de identificação da variável é feita uma consulta à tabela de símbolos.

Como foi discutido na Seção 2.3.4, em algumas linguagens, como ALGOL68 e C, pode ocorrer a declaração de variáveis no início de um bloco, sendo que o escopo dessas variáveis está limitado ao bloco onde as variáveis foram declaradas. Logo, uma variável declarada em um dado bloco vai estar indefinida fora dele.

Para tratar esse tipo de situação é que existem os conjuntos **undef** de cada nó e duas estruturas de dados auxiliares: o conjunto de "bits" (variável global) **undef** e a pilha de conjuntos de "bits" **stack_undef**. Tanto nos conjuntos **undef** dos nós como na variável global **undef**, o significado de um "bit" com o valor "1" é de que a variável associada ao "bit" está *indefinida*. A variável global **undef** é iniciada antes da análise do programa com o valor "1" em todos os "bits", indicando que, a princípio, todas as variáveis estão indefinidas. A medida em que as variáveis vão sendo declaradas, os "bits" de **undef** vão sendo "resetados", indicando que as variáveis *podem* ser definidas. Quando o analisador sintático da LI detecta um início de bloco, é empilhado em **stack_undef** o valor de **undef** antes de começar o novo bloco. Dessa maneira, está sendo salvo o conjunto de variáveis que podem ser definidas antes de começar o bloco; posteriormente, as variáveis declaradas no começo do bloco são inseridas na variável **undef**, indicando que essas variáveis podem ser definidas dentro do bloco, conjuntamente com as outras variáveis que já podiam devido a declarações mais externas ao bloco. Quando termina o bloco, o topo da pilha **stack_undef** é desempilhado e atribuído à variável **undef**, ou

seja, undef recupera o conjunto de variáveis que podiam ser definidas antes de começar o bloco que está terminado. O valor da variável undef é atribuído ao campo undef de um nó sempre que é finalizada a análise deste nó. Dessa forma, o campo undef contém o conjunto de variáveis que estão indefinidas em um dado nó, possibilitando que a seleção de caminhos ou associações seja feita somente dentro do escopo da variável.

Uma outra questão importante é a ocorrência de declarações de variáveis com o mesmo nome. Retomando a discussão da Seção 2.3.4, quando acontece a declaração, em um bloco mais interno, de uma variável com o mesmo nome de uma variável declarada externamente, a variável mais externa se torna "invisível" no bloco mais interno. Seria de se esperar que a variável mais externa fosse colocada em undef, pois estaria indefinida no bloco. No entanto, essa atitude não é necessária, nem correta, porque a variável mais externa jamais será re-definida dentro do bloco, uma vez que qualquer referência ao seu nome estará indicando a variável declarada no bloco. E ainda, é incorreto tornar inválidas as referências à variável mais externa dentro do bloco porque, para determinar os caminhos ou associações requeridos por essa variável será necessário "passar" por esse bloco para atingir um nó, ou arco, onde a referência à variável mais externa seja possível. Portanto, para implementar a re-declaração de variáveis não é exigido nenhum tratamento especial além do proposto acima.

Assim, no final da análise sintática da unidade em LI, temos os conjuntos defg, def_ref e undef construídos para cada elemento do vetor info_no, ou seja, para cada nó do GFC. Esses conjuntos, juntamente com o GFC, possibilitarão a criação dos grafo(i) utilizados na determinação dos caminhos e associações requeridos pelos critérios.

Instrumentação da Unidade em Teste

A *instrumentação*, como já foi dito, é a atividade que gera uma versão da unidade em teste preparada para a atividade de teste. Essa nova versão, denominada unidade instrumentada, possui pontas de provas (instruções de escrita) que geram um "trace" do caminho executado por um caso de teste.

As localizações das pontas de prova vão depender do comando da LI a ser instrumentado. Em última análise, a localização de uma ponta de prova depende do comando de controle de fluxo da linguagem da unidade que foi traduzida para LI. Nas Figuras 2.2, 2.3, 2.4, 2.5, 2.6 e 2.7, estão descritas as localizações das pontas de prova para monitorar os nós gerados pelos diversos comandos da LI.

Antes da inserção das pontas de prova propriamente ditas é necessário inserir alguns comandos preliminares, dependendo da linguagem da unidade em teste. Esses comandos preliminares servem, por exemplo, para definir o *procedimento* ou a *macro* que escreve o número do nó no arquivo PATH.TES e os procedimentos que

permitem a manipulação de arquivos. No caso da configuração da POKE-TOOL para a linguagem C, é inserido o comando do pré-processador que define a *macro* ponta_de_prova(x); esta *macro* é dada abaixo:

```
#define ponta_de_prova(num) \
if(printed_nodes % 10) \
  {++printed_nodes; fprintf(path," %d ",num);} \
else \
  {++printed_nodes; fprintf(path," %d\n",num);}

#include <stdio.h>
```

Os comandos acima realizam as seguintes tarefas: testa se o valor da variável printed_nodes é múltiplo de 10; em caso afirmativo, incrementa a variável printed_nodes e escreve o número do nó e o caractere "newline"; caso contrário, também incrementa a variável printed_nodes e só escreve o número do nó. Obviamente, dependendo da linguagem, não há a necessidade desse preâmbulo no arquivo da versão instrumentada; por exemplo, a linguagem Pascal não exige esses comandos necessariamente.

Existem, porém, alguns comandos fundamentais a serem inseridos, além das pontas de prova. Uma das funções desses comandos é declarar a variável que manipula o arquivo PATH.TES; a outra é abrir e fechar o arquivo PATH.TES. A declaração da variável de manipulação do arquivo e o comando de abertura do arquivo devem ser inseridos no início da unidade, antes do primeiro comando executável. O comando que fecha o arquivo PATH.TES deve ser inserido antes de todo comando *$RETURN* da LI. Para a configuração da POKE-TOOL para a linguagem C, são inseridos logo após o início da unidade os comandos:

```
FILE * path = fopen("path.tes","w");
static int printed_nodes = 0;
```

A variável path é um apontador para o arquivo PATH.TES e, no comando acima, está sendo declarada e está recebendo as informações obtidas da abertura do arquivo PATH.TES. A variável printed_nodes serve para controlar o número de nós escritos em PATH.TES e determinar quando inserir um "newline" nesse arquivo. No comando acima, printed_nodes está sendo declarada e iniciada. Para fechar o arquivo PATH.TES, é inserido antes de todo comando "return" o comando:

```
fclose(path);
```

Para controlar a instrumentação dos nós do GFC a estrutura de dados info_no possui um campo chamado to_monitor para cada nó do GFC. Este campo indica se o nó já foi monitorado ou não; inicialmente, todos os elementos de info_no são

preenchidos com o valor constante NOT_YET, que significa ainda não monitorado. À medida que vão sendo inseridas as pontas de prova, vai sendo alterado o valor do campo to_monitor para ALREADY, isto é, já monitorado. As funções que fazem a inserção das pontas de prova, antes de inserir uma ponta de prova, fazem uma consulta ao vetor info_no para saber se o nó já foi monitorado.

A tarefa da instrumentação não é somente inserir as pontas de prova, mas também gerar toda uma nova unidade. Durante a análise sintática da LI sabemos onde inserir as pontas de prova; entretanto, é necessário "preencher" a nova versão com o texto da unidade em si. Isto é feito através dos ponteiros que a LI possui para o código fonte da unidade. Quando é identificado um comando *$S* da LI, chama-se a função que monitora "statements"; esta função verifica se o nó desse *$S* já foi monitorado, caso isto não tenha ocorrido, ele altera o estado do campo to_monitor para ALREADY e insere uma *ponta de prova*; caso já tenha sido monitorado, não faz nada; em seguida, é chamado um procedimento que copia o código fonte associado a *$S* (através dos ponteiros da LI) na nova versão da unidade em teste recuperando, dessa maneira, o código relativo ao *$S*.

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

3.6.3 Geração dos Grafo(i), Caminhos, Associações e Descritores (sub-módulo des_pot_du_cam)

Este sub-módulo determina as associações e caminhos requeridos pelos critérios PU. Para realizar essas tarefas é necessário que sejam determinados os grafo(i) relativos à unidade em teste pois é a partir dos grafo(i) que são obtidos as associações, os caminhos e os descritores, que representam tanto as associações como os caminhos. Por isso, esse sub-módulo possui duas funções muito claras: inicialmente gerar os *grafo(i)* e, posteriormente, de posse dos grafo(i), gerar os caminhos, associações e descritores dos critérios PU.

Geração dos Grafo(i)

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

A utilização desses algoritmos para determinar os potenciais du-caminhos de um grafo def possui algumas vantagens. A principal delas é que se tratam de algoritmos matematicamente corretos e, portanto, garantem a corretude do resultado. Porém, eles possuem duas desvantagens: as estruturas de dados para manipulação

de potenciais du-caminhos, nesses algoritmos, podem vir a consumir muito espaço de memória; além de serem complicadas, o que torna difícil a programação de qualquer dos algoritmos.

Uma desvantagem adicional desse enfoque é que ele determina somente os requisitos de teste dos critérios todos-potenciais-du-caminhos; os demais critérios não são abordados, acarretando esforço computacional adicional para os outros dois critérios.

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

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

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

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

A seguir, serão apresentados as estruturas de dados usadas na geração dos grafo(i) e o algoritmo de geração.

Estrutura de Dados

A estrutura de dados utilizada para implementar os grafo(i) é uma lista ligada de nós, onde cada elemento dessa lista é um nó do grafo(i). Os elementos dessa primeira lista contém um campo que é um apontador para uma outra lista ligada que representa os sucessores do nó do grafo(i). Dessa maneira, o grafo(i) é representado através de várias listas; essa solução foi adotada porque, *a priori*, não sabemos quantos elementos podem existir no grafo(i) e esse número pode exceder o número de nós do GFC.

A primeira lista é composta de elementos do tipo NOGRAFOI; os elementos desse tipo são compostos dos seguintes campos:

num_grafo_i : número de identificação desse nó no grafo(i);

infosuc : este campo contém as informações a respeito do nó do grafo(i), é do tipo INFODFNO (a ser explicado mais adiante);

repetido : campo que indica se esse nó do grafo(i) tem o mesmo número no GFC de outro nó do grafo(i);

sucgrfi : apontador para os sucessores do nó no grafo(i); e

next : endereço do próximo elemento.

O campo INFODFNO contém as informações sobre o nó do grafo(i), que são:

num_no_G : o número no GFC do nó do grafo(i); e

deff : o conjunto deff desse nó do grafo(i).

Observe-se que essas duas informações identificam unicamente um nó do grafo(i).

Voltando aos elementos da lista de nós do grafo(i), notamos que o campo sucgfi é o apontador para os sucessores do nó. Este campo aponta para uma lista de elementos do tipo SUCGRAFOI; este tipo descreve as informações relativas a um sucessor de um nó do grafo(i). Abaixo, são apresentados os campos desse tipo:

num_grafo_i : idêntico ao campo do tipo NOGRAFOI;

tipo : este campo identifica se o arco constituído pelo nó do grafo(i) e o seu sucessor é *primitivo*, *herdeiro* ou p_ϕ;

no_address : este campo contém o endereço do nó sucessor na lista de nós do grafo(i) (lista de NOGRAFOI's); e

next : endereço do próximo sucessor.

Logo, o grafo(i) é representado pela lista de elementos do tipo NOGRAFOI, que são os nós do grafo(i) propriamente ditos, e por listas de sucessores (SUCGRAFOI's) associadas a cada elemento da lista de NOGRAFOI's.

Durante o desenvolvimento de um grafo(i) são obtidas todas as informações para a criação dos descritores dos três critérios PU. Essas informações estão congregadas em uma estrutura de dados chamada pdes. pdes é um vetor cujos elementos são do tipo INFODESCRITORES; cada elemento desse vetor congrega as informações obtidas durante o desenvolvimento de um grafo(i). As informações contidas em cada elemento de pdes são as seguintes:

Ni : conjunto de "bits" que contém os nós do GFC que fazem parte do grafo(i);

Nt : conjunto de "bits" que contém os nós do GFC que são nós terminais do grafo(i);

grafo_i : apontador para a lista de nós do grafo(i), ou seja, para o grafo(i);

prim_arcs : apontador para a lista de arcos primivos que fazem parte do grafo(i).

O campo prim_arcs aponta para uma lista de arcos do GFC que são primitivos ou p_ϕ no grafo(i). Um arco pode ocorrer mais de uma vez em um grafo(i), ou melhor, possuir várias imagens no grafo(i) (ver Seção 2.3.6), pois é comum ocorrer a duplicação de partes do GFC quando é criado um grafo(i). As diferentes ocorrências de um arco no grafo(i) indicam que, em termos de fluxo de dados, esses arcos são diferentes e devem ser exercitados pelos casos de teste para a satisfação dos critérios todos-potenciais-usos e todos-potenciais-usos/du (ver Seção 2.3.6). Para diferenciar entre as várias ocorrências de um arco primitivo ou p_ϕ no grafo(i) é utilizado o conjunto *deff* do nó fonte de cada ocorrência do arco. Essa informação, como já foi dito anteriormente, serve para a determinação dos descritores dos critérios todos-potenciais-usos e todos-potenciais-usos/du.

À medida que é construído um grafo(i), são colocados na lista apontada por prim_arcs todos os arcos *(i,j)* primitivos e p_ϕ, bem como os conjuntos *deff* de cada ocorrência desses arcos. prim_arcs aponta para registros do tipo ARCPRIM; ARCPRIM é definido da seguinte maneira:

arco : é um campo do tipo PAIRINT; PAIRINT é um registro que contém um par de inteiros definidos nos campos abs e coor. O nó fonte do arco é armazenado em abs e o nó alvo em coor;

deff_ptr : é o apontador para uma lista cujos elementos são conjuntos de "bits"; essa lista contém os *deff* das várias ocorrências de um arco *(i,j)*; e

next : aponta para o próximo elemento.

Algoritmo de Geração dos Grafo(i)

As estruturas de dados apresentadas acima são utilizadas pela POKE-TOOL. Antes de iniciar a descrição do algoritmo de geração dos grafo(i), vamos introduzir algumas estruturas auxiliares utilizadas nesse algoritmo.

Uma delas é o tipo de dado LIST, que nada mais é do que uma lista de elementos cujo conteúdo é do tipo INFODFNO. Mais precisamente, os campos de um elemento do tipo LIST são:

sucessor : campo do tipo INFODFNO; e

next : próximo elemento da lista.

Outra estrutura auxiliar é do tipo de dado ELEMSUC; os elementos desse tipo são definidos através dos campos:

infosuc : campo do tipo INFODFNO; e

next : apontador para uma lista de elementos do tipo LIST.

O Algoritmo

ger_grafo_i (g_suc: GRAFO; var pdes: INFODESCRITORES)

ENTRADAS:

g_suc : apontador para o GFC do programa.

SAÍDA:

pdes :vetor com as informações obtidas da geração do grafo(i).

VARIÁVEIS LOCAIS:

conodef : conjunto de nós i do GFC tal que $defg(i) \neq \phi$.

pil_cam : pilha que armazena o caminho percorrido no GFC a partir do nó i onde $defg(i) \neq \phi$. Esta pilha contém os números dos nós do GFC pertencentes ao caminho sendo percorrido.

conj_suc : pilha que auxilia no percorrimento em profundidade do GFC; seus elementos são do tipo ELEMSUC.

x : variável do tipo inteiro que contém os números dos nós i retirados de conodef.

deff : conjunto de "bits" que contém o conjunto de variáveis definidas em i mas não redefinidas no particular caminho sendo percorrido pelo algoritmo.

ref : variável do tipo ELEMSUC que armazena informações do nó i que dá origem ao grafo(i).

elemento : variável do tipo ELEMSUC que armazena os elementos retirados de conjsuc.

nosucg : variável do tipo INFODFNO que armazena informações sobre o sucessor de elemento que está sendo inserido no grafo(i).

elem_aux : variável do tipo ELEMSUC auxiliar.

i : contador de grafo(i).

fonte : variável inteira que contém o número no GFC do nó fonte do arco que será inserido no grafo(i).

alvo : variável inteira que contém o número no GFC do nó alvo do arco que será inserido no grafo(i).

VARIÁVEIS GLOBAIS:

info_no : vetor que contém, entre outras coisas, os conjuntos associados a cada nó do GFC.

FUNÇÕES e PROCEDIMENTOS chamados

det_conodef(conodef) : determina o conjunto de nós i do GFC tal que $defg(i) \neq \phi$.

ret_elem(conodef) : retira um elemento do conjunto conodef.

ini_info_descritores(pdes[i]) : inicia o elemento i do vetor pdes. Essa iniciação consiste em "resetar" os conjuntos de "bits" pdes[i].Ni e pdes[i].Nt e ajustar os campos pdes[i].grafo_i e pdes[i].prim_arcs com o valor nil.

grfi_inicializacao(pdes[i], ref) : esta função insere no grafo(i) apontado por pdes[i].grafo_i o nó definido por ref.infosuc.num_no_G e ref.infosuc.deff que nada mais é do que o nó i e o conjunto $defg(i)$.

push(elem, stack) : empilha elem em stack.

pop(stack) : retira e retorna o elemento do topo da pilha stack.

set_bit(num, conj) : ajusta com o valor "1" o "bit" de número num no conjunto de "bits" conj.

inter_bit(conj1,conj2) : faz a interseção entre dois conjuntos de "bits" e retorna o conjunto resultado.

neg_bit(conj) : faz a negação dos elementos do conjunto de "bits" conj e retorna o novo conjunto obtido.

sub_bit(conj1,conj2) : subtrai o conjunto de "bits" conj2 de conj1 retornando o conjunto obtido.

e_vazio_bit(conj) : retorna true se todos os "bits" de conj forem iguais a "0" e false, caso contrário.

atualiza_pilcam(elem) : atualiza o apontador da pilha pil_cam de forma que o topo da pilha seja sempre igual a elem.

head(elemento.apont) : retira e retorna um elemento da lista de elementos do tipo LIST apontado por elemento.apont.

e_primitivo(fonte, alvo, g_suc) : verifica no GFC apontado por g_suc se o arco definido por (fonte, alvo) é um primitivo de fluxo de dados. e_primitivo retorna true se for primitivo e false, caso contrário.

insere_grafo_i(elemento, nosucg, pdes[i].grafo_i, tipo) : esta função insere um novo arco no grafo(i), criando um novo nó; este arco tem o nó fonte definido por elemento.infosuc e o nó alvo por nosucg. pdes[i].grafo_i é o ponteiro para o início da lista de nós do grafo(i) e tipo indica se o nó alvo (que está sendo inserido no grafo(i)) possui número no GFC idêntico ao de algum outro nó do grafo(i); nesta situação, o valor tipo é REP; caso não exista nó com número igual, o valor de tipo é N_REP.

ins_arc(pdes[i].prim_arcs, alvo, fonte, deff) : insere na lista de arcos primitivos apontada por pdes[i].prim_arcs o arco definido por (fonte, alvo) (se ele já não estiver na lista) e associa o conjunto de "bits" deff ao arco (fonte, alvo).

existe_no_igual(nosucg,pdes[i].grafo_i) : retorna true se existir um nó no grafo(i) apontado por pdes[i].grafo_i com número no GFC igual a nosucg.num_no_G e deff igual a nosucg.deff; e false, caso contrário.

liga_grafo_i(elemento, nosucg, pdes[i].grafo_i) : faz a ligação do nó definido por elemento.infosuc para o nó definido por nosucg no grafo(i) apontado por pdes[i].grafo_i.

lista_sucessores(x, g_suc) : cria, a partir do GFC, uma lista de sucessores de x cujos elementos são do tipo LIST. Os elementos da lista são iniciados com os números dos sucessores no GFC e seus respectivos conjuntos defg(i).

faca_arco_pfi(elemento, nosucg, pdes[i].grafo_i) : altera o estado do arco definido por (elemento.infosuc.num_no_G, nosucg.num_no_G) do grafo(i) apontado por pdes[i].grafo_i de primitivo ou herdeiro para p_ϕ.

```
begin
/* determina os conjuntos de no's do program que possui
   definicao de varia'veis */
conodef = det_conodef(conodef);
i = 1;
while conodef <> nil do
  begin
  x = ret_elem(conodef);
```

```
/* inicia ref com as informacoes do no' x */
ref.infosuc.num_no_G = x;
ref.infosuc.def = uniao_bit(info_no[x].defgi,info_no[x].def_ref);
ref.apont = lista_sucessores(x, g_suc);
ini_info_descritores(pdes[i]);
grfi_inicializacao(pdes[i], ref);
push(ref, conjsuc);
push(x, pil_cam);
while conjsuc <> nil do
  begin
  elemento = pop(conjsuc);
  fonte = elemento.infosuc.num_no_G;
  set_bit(fonte, pdes[i].Ni);
  atualiza_pilcam(fonte, pil_cam);
  deff = elemento.infosuc.deff;
  if elemento.apont <> nil then
    nosucg = head(elemento.apont);
  if elemento.apont <> nil then
    push(elemento, conjsuc);
  alvo = nosucg.num_no_G;
  if e_primitivo(fonte,alvo,g_suc) then
    ins_arc(pdes[i].prim_arcs,fonte,alvo,deff);
  deff = sub_bit(
deff,inter_bit(deff,inter_bit(info_no[alvo].defgi,neg_bit(info_no[i].undef)))
  nosucg.deff = deff;
  if e_vazio_bit(deff) then
    begin
    if pertence_pil_cam(alvo, pil_cam) then
      begin
      set_bit(alvo,pdes[i].Nt);
      insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,REP);
      faca_arco_pfi(elemento, nosucg, pdes[i].grafo_i);
      ins_arc(pdes[i].prim_arcs,alvo,fonte,elemento.infosuc.deff);
      end
    else
      if existe_no_igual(nosucg, pdes[i].grafo_i) then
        liga_grafo_i(elemento, nosucg, pdes[i].grafo_i);
      else
        begin
        insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,N_REP);
        push(alvo,pil_cam);
```

```
          elem_aux.infosuc = nosucg;
          elem_aux.apont = lista_sucessores(nosucg.num_no_G,g_suc);
          if elem_aux.apont <> nil then
            push(elem_aux,conjsuc);
          else
            begin
            set_bit(alvo,pdes[i].Ni);
            set_bit(alvo,pdes[i].Nt);
            end
          end
      end
    else  /* deff = vazio */
      begin
      if pertence_pil_cam(alvo, pil_cam) then
        begin
        set_bit(alvo,pdes[i].Nt);
        insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,REP);
        faca_arco_pfi(elemento, nosucg, pdes[i].grafo_i);
        ins_arc(pdes[i].prim_arcs,alvo,fonte,elemento.infosuc.deff);
        end
      else
        if existe_no_igual(nosucg, pdes[i].grafo_i) then
          begin
          set_bit(alvo,pdes[i].Nt);
          liga_grafo_i(elemento, nosucg, pdes[i].grafo_i);
          faca_arco_pfi(elemento, nosucg, pdes[i].grafo_i);
          ins_arc(pdes[i].prim_arcs,alvo,fonte,elemento.infosuc.deff);
          end
        else
          begin
          set_bit(alvo,pdes[i].Nt);
          insere_grafo_i(elemento,nosucg,pdes[i].grafo_i,N_REP);
          faca_arco_pfi(elemento, nosucg, pdes[i].grafo_i);
          ins_arc(pdes[i].prim_arcs,alvo,fonte,elemento.infosuc.deff);
          end
        end
      end
    end
  i := i + 1;
  end
end
```

Geração dos Descritores, Caminhos e Associações

Terminado o processo de geração do grafo(i), tem-se a estrutura de dados pdes preenchida com todas as informações necessárias para a geração dos descritores, caminhos e associações requeridas pelos critérios PU.

Os descritores gerados para o critérios todos-potenciais-du-caminhos, todos-potenciais-usos e todos-potenciais-usos/du são armazenados nos arquivos DES_PDU.TES, DES_PU.TES e DES_PUDU.TES, respectivamente. Os caminhos requeridos pelos critérios todos-potenciais-usos são armazenados no arquivo PDUPATHS.TES e as associações requeridas pelos critérios todos-potenciais-usos e todos-potenciais-usos/du são armazenados no arquivo PUASSOC.TES. No Apêndice A, temos os arquivos descritos acima gerados para o exemplo ENTAB.C.

Descritores e Caminhos do Critério Todos-Potenciais-Du-Caminhos

Para gerar os descritores e caminhos requeridos pelo critério todos-potenciais-du-caminhos, o grafo(i) é percorrido em profundidade. *Percorrer em profundidade* consiste em fazer uma busca "depth-first" no grafo(i), armazenando, em uma *pilha de nós*, os nós visitados e, em uma *pilha de arcos*, os arcos primitivos e herdeiros encontrados no grafo(i) percorrido. É bom lembrar que os nós e arcos do grafo(i) percorridos são armazenados nas pilhas acima através dos seus números e pares de números no GFC.

Quando é atingido um nó terminal do grafo(i) é feita a geração do caminho e do descritor. Para gerar o caminho, basta escrever os nós contidos na *pilha de nós* em ordem inversa (i.e. o nó do topo da pilha é o último) no arquivo PDUPATHS.TES. Para gerar o descritor também deve-se processar os arcos contidos na *pilha de arcos* na ordem inversa, porém tomando-se alguns cuidados. Inicialmente, escreve-se a cadeia de caracteres "N* i", depois são escritos os arcos primitivos *(l,m)* da seguinte maneira:

(1) se $i = l$, então, escreve-se somente o nó m;

(2) se o último arco primitivo foi o arco (k,j) e $j = l$, escreve-se somente o nó m; e

(3) se o número de arcos herdeiro que precedem o arco (l,m) for maior ou igual a dois, então, antes de escrever os nós l e m, é escrita a cadeia "Nlf* "; caso contrário, escrevem-se os nós l e m.

Descritores e Caminhos dos Critérios Todos-Potenciais-usos e Todos-Potenciais-usos/du

A determinação das associações e dos descritores dos critérios todos-potenciais-usos e todos-potenciais-usos/du é feita através da lista de arcos apontada por

pdes[i].prim_arcs. Nesta lista tem-se os arcos primitivos *(j,k)* (em termos de nós do GFC) mais os conjuntos $deff(f_\pi(j))$, para cada ocorrência do arco *(j,k)* no grafo(i).

Para cada ocorrência de um arco primitivo, é criada uma associação */i, (j,k), $deff(f_\pi(j))$/* e seus respectivos descritores em cada critério. Entretanto, como a especificação de uma associação é a mesma nos dois critérios, é criado um único arquivo (PUASSOC.TES) com as associações requeridas.

Então, para gerar as associações requeridas pelos dois critérios basta tomar o nó i que gerou o grafo(i), os arcos primitivos *(j,k)* da lista contida em pdes[i] e, para cada ocorrência do arco *(j,k)* especificada pelos conjuntos $deff(f_\pi(j))$, escrever a associação

$$[i, (j,k), \{v_1, v_2, \ldots, v_n\}]$$

no arquivo PUASSOC.TES, onde $v_1, v_2, \ldots, v_n$ são variáveis contidas no conjunto $deff(f_\pi(j))$. Para obter as variáveis $v_1, v_2, \ldots, v_n$ propriamente ditas, é feita uma consulta a uma estrutura de dados que associa o número de identificação de uma variável à sua cadeia de caracteres, pois na lista apontada por pdes[i].prim_arcs, o conjunto $deff(f_\pi(j))$ está codificado na forma de um conjunto de "bits".

A geração do descritor de uma associação definida por um arco *(j,k)* e um conjunto $deff(f_\pi(j))$ é feita da seguinte maneira:

(1) escreve-se a cadeia de caracteres "N* i Nnv* j [Nnv* j]* k", se $i \neq j$, ou "N* i [Nnv* i]* k", se $i = j$, para o critério todos-potenciais-usos;

(2) escreve-se a cadeia de caracteres "N* i Nnvlf* j k", se $i \neq j$, ou "N* i k", se $i = j$, para o critério todos-potenciais-usos/du;

(3) calcula-se o conjunto N_{nv} (ou N_{nvlf} no caso critério todos-potenciais-usos/du) através da operação

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

onde $N_v = \{n \in N \mid v \in defg(n)\}$.

(4) escreve-se o conjunto N_{nv} (ou N_{nvlf}) no arquivo de descritores associado a esse descritor.

No Apêndice A são apresentados os arquivos PDUPATHS.TES, PUASSOC.-TES, DES_PDU.TES, DES_PU.TES e DES_PUDU.TES gerados pela análise estática de um programa real.

3.7 Módulo Avaliador

Conforme definido na Seção 2.3.6, os descritores dos caminhos e associações requeridos pelos critérios PU são autômatos finitos que devem ser satisfeitos para que o caminho ou a associação seja considerada exercitada. Para exercitar um descritor de caminho ou associação é necessário que exista um *caminho*, dentro de um caminho completo, que realize as transições do *estado inicial* até o *estado final* do autômato.

Considere, por exemplo, o caminho (2, 3, 7, 9, 10, 12, 13, 14, 15) requerido pelo critério todos-potenciais-du-caminhos no exemplo contido no Apêndice A. Este caminho começa no nó 2 e é descrito pela seguinte seqüencia de arcos primitivos:

$$(10,12)(14,15).$$

O descritor gerado para este caminho é dado por

```
N* 2 Nlf* 10 12 Nlf* 14 15.
```

Então, vê-se que é necessário um caso de teste cujo caminho completo executado provoque a transição do estado s_1 (estado inicial) até o estado s_f (estado final) para a satisfação do autômato e, consequentemente, do caminho requerido pelo critério todos-potenciais-du-caminhos.

No descritor, N é o conjunto de todos os nós do GFC e N_{lf} é o conjunto $N_i/1$, ou seja, o conjunto N_i com características dinâmicas. O símbolo "*" significa que qualquer elemento dos conjuntos N e N_{lf} poderão ocorrer a menos do elemento correspondente ao número imediatamente seguinte a `N*` ou `Nlf*`, quando, então, ocorre uma transição do autômato. O conjunto N_{lf} tem uma particularidade; ele é dinâmico, isto é, ele é alterado durante o processo de avaliação. O subscrito *lf* é a abreviatura para "loop-free" e indica que os elementos desse conjunto só podem ocorrer uma vez durante as transições do autômato correspondente. Digamos que o autômato esteja em um estado s_i e ocorra um nó q tal que $q \in N_{lf}$; neste caso, vai ocorrer a transição s_i para s_i, mas o nó q será retirado de N_{lf}, de tal maneira que, se ocorrer novamente o nó q, a transição s_i para s_i não se realizará porque q não pertence mais a N_{lf}, e o autômato será reiniciado. Dessa forma, com o conjunto N_{lf} dinâmico, garante-se que o caminho aceito pelo descritor é um potencial du-caminho.

Então, vê-se que é necessário um caso de teste cujo caminho completo executado provoque a transição do estado s_1 (estado inicial) até o estado s_f (estado final) para a satisfação do autômato e, consequentemente, do caminho requerido pelo critério todos-potenciais-du-caminhos.

Para descrever, por exemplo, uma associação */2,(9,14),{newcol}/* requerida pelo critério todos-potenciais-usos é utilizado o seguinte descritor

```
N* 2 Nnv* 9 [ Nnv* 9 ]* 14.
```

N_{nv} é o conjunto de nós do grafo(i), com exceção daqueles nós onde ocorreram definições de v. Note-se que se a associação fosse $/2,(9,14),\{newcol,w\}/$, o conjunto N_{nv} seria o conjunto de nós do grafo(i) reduzido daqueles nós com definição de *newcol* e w.

Analogamente a um potencial du-caminho, é necessário que o caminho executado pelo caso de teste realize as transições de s_1 até s_f para que a associação seja satisfeita. Note-se que neste caso não ocorre redução do conjunto N_{nv} porque o critério todos-potenciais-usos permite a repetição de qualquer nó onde não haja repetição de *newcol*.

Os descritores do critério todos-potenciais-usos/du são mais simples. Considere a associação $/2,(9,14),\{newcol\}/$; o descritor é

```
N* 2 Nnvlf* 9 14.
```

O conjunto N_{nvlf} é, inicialmente, igual a N_{nv}, porém é alterado dinamicamente como o conjunto N_{lf}; a cada ocorrência de um nó pertencente a N_{nvlf}, ele é retirado do conjunto para garantir que somente potenciais du-caminhos satisfaçam as associações.

3.7.1 Implementação da Avaliação na POKE-TOOL

O avaliador da POKE-TOOL recebe como entradas um arquivo com os caminhos ou associações requeridos pelo critério selecionado (PDUPATHS.TES, ou PUASSOC.TES), um arquivo com os descritores desse critério (DES_PDU.TES, DES_PU.TES ou DES_PUDU.TES) e o arquivo que contém o caminho executado pelo caso de teste (PATH.TES). Como saída, o avaliador gera um arquivo com os caminhos ou associações que restam ser executados (PDUOUTPUT.TES, PUOUTPUT.TES ou PUDUOUTPUT.TES), um arquivo com os caminhos ou associações efetivamente executados (EXEC_PDU.TES, EXEC_PU.TES ou EXEC_PUDU.TES) e um arquivo que contém a identificação dos descritores exercitados pelo caso de teste (PDUHIS.TES, PUHIS.TES ou PUDUHIS.TES). Estes últimos arquivos servem como um histórico da avaliação de um dado critério, pois a avaliação de um novo caso de teste vai utilizar estes arquivos para saber quais os descritores que já foram satisfeitos em avaliações anteriores. Os arquivos que contêm o resultado da avaliação (por exemplo, os arquivos PUOUTPUT.TES e EXEC_PU.TES para o critério todos-potenciais-usos) apresentam dois percentuais de *cobertura* para o critério selecionado, conforme descrito na Seção 2.3.6.

Este módulo é um programa executável constituído de duas partes: a primeira parte é chamada *iniciação* dos autômatos e a segunda é a *avaliação* dos autômatos.

O módulo avaliador primeiramente faz a iniciação dos autômatos finitos através do arquivo de descritores. Esta iniciação é dependente do critério, mas

consiste, basicamente, no estabelecimento dos autômatos na memória. Com os autômatos na memória é possível proceder à análise dos autômatos frente ao caminho executado. O algoritmo de avaliação faz a análise de todos os autômatos simultâneamente através de uma única passada pelo arquivo com o "trace" do caso de teste. É importante observar que o *algoritmo de avaliação* trata descritores, não associações ou caminhos.

Depois de dar uma passada pelo arquivo PATH.TES, têm-se alguns autômatos no estado final e outros não. Aqueles que estão no estado final indicam que os caminhos ou associações associados a eles foram executados pelo caso de teste. De posse dessas informações, são gerados o arquivo resultado e o que contém o histórico das avaliações.

A seguir, vão ser descritos a estrutura de dados que representa os autômatos, o processo de iniciação e o algoritmo de avaliação dos critérios PU; este algoritmo é uma extensão do algoritmo de avaliação apresentado em [FRA87].

Estrutura de Dados

Os autômatos são obtidos a partir dos descritores do critério selecionado e são organizados na forma de uma lista ligada. Cada elemento da lista designa um autômato obtido de um descritor; os campos contidos nesses elementos são os seguintes.

i : este campo identifica o nó *i* de onde foi obtido o grafo(i) e os descritores.

n_path : é o número de identificação associado ao autômato. Este número de identificação é igual ao número do caminho ou associação nos arquivos PDU-PATHS.TES, PUASSOC.TES, DES_PDU.TES, DES_PU.TES e DES_PU-DU.TES.

exp_regular : é o apontador para o descritor do caminho ou associação. Ele aponta para a cadeia de caracteres que contém o descritor.

pos_corrente : é o apontador que indica em que ponto do descritor está a avaliação, ou melhor, indica em que estado o autômato se encontra no processo de avaliação.

Ni : é um conjunto de "bits" que contém os nós que fazem parte do grafo(i) (conjunto N_i) quando estão sendo avaliados os autômatos do critério todos-potenciais-du-caminhos. Para os critérios todos-potenciais-usos e todos-potenciais-usos/du, o campo Ni é utilizando para armazenar os conjuntos N_{nv} e N_{nvlf}, respectivamente. Esse campo não é alterado durante o processo de avaliação.

Nnv : este campo é um conjunto de "bits" e é utilizado para armazenar o conjunto N_{nv} quando estão sendo avaliados os autômatos do critério todos-potenciais-usos. Para o critério todos-potenciais-usos/du e todos-potenciais-du-caminhos, o campo Nnv é utilizado para os conjuntos N_{nvlf} e N_{lf}, respectivamente. Nesses dois últimos casos, o campo Nnv é alterado no decorrer da avaliação.

estado : este campo indica o estado do autômato durante o processo de avaliação. Ele pode assumir três estados: Q1, Q2 e Q3. Q1 indica que o processo de avaliação está ainda transcorrendo; Q2 indica que o processo de avaliação falhou, ou seja, o autômato não *aceitou* o caminho executado; e Q3 indica que o processo de avaliação foi bem sucedido e, portanto, o caminho do caso de teste exercitou o caminho ou associação.

next : contém o endereço do próximo elemento da lista de autômatos; o último elemento tem next igual a nil.

Os elementos da lista ligada de autômatos são do tipo AUTOMATO e os apontadores para esse tipo são do tipo PAUTOMATO.

Iniciação dos Autômatos

O processo de iniciação dos autômatos varia muito pouco de critério para critério. Para os critérios todos-potenciais-usos e todos-potenciais-usos/du, o processo de iniciação é idêntico; para o critério todos-potenciais-du-caminhos, a diferença é mínima em relação aos outros dois. É fácil notar, a partir dos arquivos DES_PDU.TES, DES_PU.TES e DES_PUDU.TES do exemplo contido no Apêndice A, que esses arquivos contêm toda a informação necessária para a avaliação. Os descritores nesses arquivos estão agrupados por grafo(i), possuem um número de identificação e os conjuntos N, N_i, N_{nv} e N_{nvlf} estão especificados. A diferença entre os processos de iniciação reside nos conjuntos N_{nv} e N_{nvlf}. Para os critérios todos-potenciais-usos e todos-potenciais-usos/du, a cada descritor está associado um conjunto N_{nv} ou N_{nvlf} que será atribuído aos campos Nnv e Ni do elemento da lista de autômatos desse descritor; para o critério todos-potenciais-du-caminhos, os campos Nnv e Ni receberão o conjunto N_i (geral para todos os descritores de um grafo(i)) que é o valor inicial do conjunto N_{lf}. Logo, a iniciação dos autômatos para o critério todos-potenciais-du-caminhos é mais simples, uma vez que o conjunto N_i é lido para cada grupo de descritores de um grafo(i) e o conjunto N_{nv} (ou N_{nvlf}) é lido obrigatoriamente para cada descritor.

Terminada a iniciação dos autômatos, tem-se uma lista ligada cujos elementos possuem os respectivos campos com o valores: i, com o número i do grafo(i);

n_path, com o número de identificação obtido do arquivo de descritores; **exp_regular** e **pos_corrente**, apontando para o início da cadeia de caracteres que representa o descritor; **Ni**, com o valor dos conjuntos N_i, N_{nv} ou N_{nvlf}, dependendo do critério; **Nnv**, com o valor de N_i, N_{nv} ou N_{nvlf}, dependendo do critério; **estado**, com o valor Q1[4]; e **next**, com o endereço do próximo elemento na lista de autômatos.

Algoritmo de Avaliação

O algoritmo de avaliação simula os autômatos finitos representados pelos descritores. A idéia por trás desse algoritmo é a seguinte. O algoritmo lê um nó do arquivo PATH.TES e, de posse desse nó, ele percorre a lista de autômatos realizando as transições. Assim, o resultado a respeito da aceitação ou não do caminho pelos autômatos só será sabido quando for lido o último nó; este resultado é obtido para todos os autômatos de uma vez.

Para entender melhor as transições, considere o descritor "N* 7 Nlf* 8 9" de um potencial du-caminho. O diagrama de estados do autômato relativo a esse descritor é mostrado na Figura 3.5. No início, os campos **exp_regular** e **pos_corrente** apontam para a cadeia toda do descritor "N* 7 Nlf* 8 9", ou seja, o autômato está no estado 1. O algoritmo separa, então, o primeiro átomo da cadeia apontada por **pos_corrente**, neste caso "N*"; dessa maneira, o algoritmo "sabe" que ele está em um estado onde duas transições são possíveis: se o nó lido for igual a 7, o autômato muda para o estado 2 e **pos_corrente** passa a apontar para a cadeia "Nlf* 8 9" ou, se o nó lido for diferente de 7 mas pertencer a N, o autômato não muda de estado e **pos_corrente** continua apontando para "N* 7 Nlf 8 9". Digamos que a primeira situação tenha ocorrido; **pos_corrente** passa a apontar para "Nlf* 8 9". Para continuar a análise, o algoritmo pega um novo nó e separa o primeiro átomo da cadeia apontada por **pos_corrente**, que é "Nlf*" neste caso. Novamente, o algoritmo "sabe" que o autômato deve permanecer no estado em que está, se o nó lido é diferente do átomo seguinte a "Nlf*" mas pertencer a N_{lf}, ou mudar de estado, se o nó lido for igual ao átomo seguinte. Todavia, existe ainda uma terceira possibilidade: o nó lido ser diferente do átomo seguinte a "Nlf*" e não pertencer a N_{lf}. Nesta situação não há transição possível; portanto, ocorreu uma falha no processo de avaliação do caminho para esse autômato e o campo **estado** é mudado para o valor Q2. Entretanto, se o nó 7 ocorrer novamente no caminho executado, este autômato é "resetado" e pode ser mais uma vez satisfeito; o campo **estado** recebe o valor Q1 e **pos_corrente** passa a apontar para "Nlf 8 9" e N_{lf} é reiniciado.

O último nó do arquivo PATH.TES é o nó 0. Este nó foi adicionado somente

[4]O valor do campo **estado** pode ser igual a Q3, caso o número de identificação do descritor esteja no arquivo que contém o histórico de avaliações do critério.

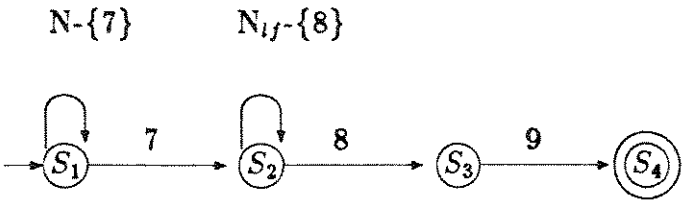


Figura 3.5: Autômato finito associado ao descritor "N* 7 Nlf* 8 9".

para indicar o final da cadeia de entrada (caminho executado). Quando o algoritmo lê um 0, ele sabe que terminou o caminho executado e, então, passa a verificar quais os autômatos que chegaram ao estado final. Um autômato chegou ao estado final quando o campo **pos_corrente** aponta para o final da cadeia do descritor e o campo **estado** é Q1. Se essas condições se verificam, o campo **estado** é mudado para Q3. Assim, depois de uma passada pelo arquivo PATH.TES, temos os autômatos em dois estados possíveis Q2 ou Q3, e é possível criar os arquivos resultantes da avaliação.

A descrição apresentada acima foi para um autômato associado a um descritor de um potencial-du-caminho. A descrição para as associações dos critérios todos-potenciais-usos e todos-potenciais-usos/du é semelhante e pode ser simulada através do algoritmo de avaliação apresentado abaixo. Este algoritmo é chamado de **avaliador**.

O Algoritmo de Avaliação

avaliador(automatos_inicio: PAUTOMATO; N: bitvector; criterio: **char**)

FUNÇÃO:

Avaliar os autômatos finitos associados aos descritores de um critério PU.

ENTRADAS:

automatos_inicio : este parâmetro é o apontador para a lista ligada de autômatos; seu tipo é PAUTOMATO, isto é, apontador para um registro do tipo AUTOMATO.

N : é um conjunto de "bits" que contém os números dos nós pertencentes ao GFC.

criterio : indica qual critério está sendo avaliado.

SAÍDA:

Autômatos avaliados, ou seja, com o campo estado ajustado com Q2 ou Q3.

VARIÁVEIS LOCAIS:

literal : vetor de caracteres que contém o primeiro átomo da cadeia apontada pelo campo **pos_corrente**.

lit_aux : vetor de caracteres que contém o segundo átomo da cadeia apontada pelo campo **pos_corrente**.

node : variável inteira que vai conter o nó lido do arquivo PATH.TES.

atual_automato : apontador para um registro do tipo AUTOMATO.

executed_path : apontador para o arquivo PATH.TES.

ptr_aux : apontador para cadeia de caracteres auxiliar.

CONSTANTES DEFINIDAS:

PDU = '1'

PU = '2'

PUDU = '3'

FUNÇÕES e PROCEDIMENTOS chamados:

peg_tok(vet1,vet2) : coloca em *vet1* o primeiro átomo da cadeia de caracteres *vet2*.

peg_next_tok(vet1,vet2) : coloca em *vet1* o segundo átomo da cadeia de caracteres *vet2*.

apont_next_tok(pvet) : *pvet* é um apontador para uma cadeia de caracteres *vet*. *apont_next_tok* faz *pvet* apontar para a cadeia de caracteres *vet* a partir do segundo átomo, ignorando o primeiro átomo.

apont_prev_tok(pvet) : faz *pvet* apontar para a cadeia de caracteres *vet* a partir do átomo anterior ao primeiro átomo corrente.

jump_tok(pvet) : faz *pvet* apontar para o terceiro átomo da cadeia de caracteres *vet*.

e_numero(pvet) : retorna true se *pvet* apontar para uma seqüência de caracteres que determinem um número; false, caso contrário.

fim_exp_reg(pvet) : retorna true se *pvet* apontar para uma cadeia vazia; false, caso contrário.

readno(var1,file) : lê um inteiro do arquivo apontado por *file* e coloca em *var1*.

erro("cadeia") : imprime a cadeia de caracteres "cadeia" e termina o programa.

atoi(pvet) : converte a cadeia de caracteres ASCII apontada por *pvet* para inteiro, retornando o inteiro.

```
begin
readno(node, executed_path); /* le um no' e coloca em node */
while node <> 0 do
  begin
  atual_automato = automatos_inicio;
  while atual_automato <> nil do
    begin
    if atual_automato^estado = Q1 then
      begin /* esta' em processo de reconhecimento */
      peg_tok(literal,atual_automato^pos_corrente);
      if literal = "N*" then
        begin /* e' igual a N* */
        peg_next_tok(lit_aux,atual_automato^pos_corrente);
        if e_numero(lit_aux) then
          begin
          numero = atoi(lit_aux);
          if node = numero then
          /* reconheceu o no', caminha no descritor */
            jump_tok(atual_automato^pos_corrente);
          else
            if not test_bit(node, N) then
            /* na~o reconheceu o caminho ate' o momento */
              atual_automato^estado = Q2;
          end
        end
      else
        error("* * Erro Fatal: Descritor incorreto");
      if literal = "Nnvlf*" then
        begin /* e' igual a Nnvlf* */
        peg_next_tok(lit_aux,atual_automato^pos_corrente);
```

```
  if e_numero(lit_aux) then
    begin
    numero = atoi(lit_aux);
    if node = numero then
    /* reconheceu o no', caminha no descritor */
      jump_tok(atual_automato^pos_corrente);
    else
      if not test_bit(node, atual_automato^Nnv) then
      /* na~o reconheceu o caminho ate' o momento */
        atual_automato^estado = Q2;
    end
  end
else
  error("* * Erro Fatal: Descritor incorreto");
if literal = "Nlf" then
  begin
  /* Identico ao corpo do if quando
     literal e' igual a "Nnvlf" */
  end
if literal = "Nnv"  then
  begin
  /* Identico ao corpo do if quando
     literal e' igual a "Nnvlf" */
  end
if e_numero(literal) then
  begin /* e' nu'mero */
  numero = atoi(literal);
  if numero = node then
  /* reconheceu o no', caminha no descritor */
    apont_next_tok(atual_automato^pos_corrente);
  else
    atual_automato^estado = Q2;
  end
if literal = "[" then
  begin /* e' igual a [ */
  /* faz ptr_aux apontar para o a'tomo "]*" */
  ptr_aux = atual_automato^pos_corrente;
  apont_next_tok(jump_tok(ptr_aux));
  peg_next_tok(lit_aux,ptr_aux);
  if e_numero(lit_aux) then
    begin
```

```
        if node = numero then
          begin
          /* reconheceu o no', caminha no descritor */
          atual_automato^pos_corrente = ptr_aux;
          jump_tok(atual_automato^pos_corrente);
          end
        else
          if not test_bit(node, atual_automato^Nnv) then
          /* na~o reconheceu o caminho ate' o momento */
            atual_automato^estado = Q2;
          else
            apont_next_tok(atual_automato^pos_corrente);
        end
      end
    else
      error("* * Erro Fatal: Descritor incorreto");
    if literal = "]*" then
      begin /* e' igual a ]* */
      peg_next_tok(lit_aux,atual_automato^pos_corrente);
      if e_numero(lit_aux) then
        begin
        numero = atoi(lit_aux);
        if node = numero then
        /* reconheceu o no', caminha no descritor */
          jump_tok(atual_automato^pos_corrente);
        else
          if not test_bit(node, atual_automato^Nnv) then
          /* na~o reconheceu o caminho ate' o momento */
            atual_automato^estado = Q2;
          else
            apont_prev_tok(apont_prev(atual_automato^pos_corrente));
        end
      end
    else
      error("* * Erro Fatal: Descritor incorreto");
  end
if atual_automto^estado = Q2 and atual_automato^i = node then
  begin   /* reseta auto^mato */
  atual_automto^estado = Q1;
  atual_automato^pos_corrente = atual_automto^exp_regular;
  jump_tok(atual_automto^pos_corrente);
```

```
    /* se o crite'rio escolhido na~o e' todos-pot-usos,
       enta~o recupera o conjunto Nnvlf ou Nlf */
    if criterio <> PU then
      atual_automato^Nnv = atual_automato^Ni;
    end

   /* se o crite'rio na~o e' todos-pot-usos e o conjunto
      testado foi Nnvlf ou Nlf, enta~o retira o no' do conjunto */

  if criterio <> PU and literal <> "N*" and atual_automato <> nil then
    reset_bit(no, atual_automato^Nnv);
  atual_automato = atual_automato^next;
  end

/* terminou de avaliar caso de teste, agora ajusta auto^matos */

atual_automato = automoatos_inicio;
while atual_automato <> nil do
  begin
  if atual_automato^estado <> Q2 then
   if fim_exp_reg(atual_automato^pos_corrente) then
     atual_automato^estado = Q3;
   else
     atual_automato = Q2;
  atual_automato = atual_automato^next;
  end
end
```

No Apêndice A são apresentados os arquivos PDUOUTPUT.TES, PUOUTPUT.TES, PUDUOUTPUT.TES, EXEC_PDU.TES, EXEC_PU.TES e EXEC_PUDU.TES obtidos após a avaliação de um conjunto de casos de teste executados para o exemplo ENTAB.C.

Capítulo 4

Aspectos de Configuração da Ferramenta POKE-TOOL

4.1 Necessidade de uma Ferramenta Multilinguagem

Hoje em dia existe uma enorme gama de linguagens sendo utilizadas na implementação dos mais diversos sistemas de software. Nessa torre de Babel, estabelecida pelas várias linguagens, encontramos linguagens mais antigas como o FORTRAN e o COBOL que ainda possuem um público muito fiel, principalmente nas comunidades científicas e comercial; temos também as linguagens derivadas do ALGOL 60 e ALGOL68 que vêm sendo muito utilizadas no ensino de programação (e.g.: Pascal) e na programação de sistemas (e.g.: C); mais recentemente (décadas de 70 e 80), surgiram linguagens que incorporaram conceitos de *tipos abstratos de dados* e *orientação a objetos* (e.g.: Ada, Modula-2, C++, etc) e que têm conseguido cativar um grande número de programadores. A relação apresentada acima se restringe apenas às linguagens do paradigma de programação procedural; se fôssemos considerar os demais paradigmas, a lista se estenderia mais ainda. Porém, mesmo limitando-se às linguagens procedurais, a lista é grande e já aponta para uma característica necessiária nas ferramentas de apoio ao teste de unidades: prover facilidades para o suporte de diversas linguagens de programação. A POKE-TOOL provê essas facilidades.

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

mecanismo de *Avaliação* utilizado são completamente independentes da linguagem fonte.

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

A POKE-TOOL utiliza o enfoque genérico. A razão para essa opção foi privilegiar a reutilização de código (e esforço) em detrimento da otimização. Com esse enfoque, estamos reutilizando também parte do código usado nos módulos dependentes da linguagem e reutilizando outras ferramentas, como descrito nos módulos **li** e **chanomat**.

Não obstante a existência de soluções otimizadas e, portanto, melhores sob o ponto de vista de eficiência, para o enfoque específico, os algoritmos genéricos de análise sintática e léxica utilizados pela POKE-TOOL têm uma complexidade ($n \log n$, onde n é o comprimento da cadeia de entrada) que não constitue um gargalo no desempenho da ferramenta; este fato foi verificado, na prática, através da utilização da ferramenta na condução de um "benchmark" [MAL91a].

Neste capítulo, iremos discutir os apectos de configuração da ferramenta POKE-TOOL, apresentando os pontos da ferramenta que são dependentes da linguagem de programação e a forma de atuar na mesma para obter a configuração para uma nova linguagem. Ainda, é apresentado um exemplo de utilização da presente configuração da ferramenta.

4.1.1 Aspectos Dependentes da Linguagem

As atividades da POKE-TOOL relacionadas com a análise estática do código fonte são intrinsecamente dependentes da linguagem fonte da unidade.

Retomando a Figura 2.1, observamos que, das funções discriminadas, as seguintes estão relacionadas com a análise do código fonte: *Grafo de Fluxo de Controle*, *Extensão do Grafo de Fluxo de Controle* e *Instrumentação*.

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

A *Extensão do Grafo de Fluxo de Controle* consiste em gerar o grafo def. Para gerar o grafo def é necessário que esteja definido o *Modelo de Dados* para a lingua-

gem da unidade em teste. Para a definição do Modelo de Dados existem diretrizes gerais (ver Seção 2.3.4), mas o modelo em si é específico para uma dada linguagem. Adicionalmente, essa função determina quais as variáveis que são *definidas* em um dado nó, o que implica na análise de cada comando que compõe um nó do GFC. Ou seja, essa análise é dependente do código fonte e pode ser feita segundo os dois enfoques descritos em 4.1.

A *Instrumentação* também é uma função cuja dependência da linguagem de programação se restringe basicamente ao mapeamento para a LI, pois a localização das pontas de prova colocadas na versão modificada depende dos comandos da LI (ver Seção 2.3.3); ou seja, uma vez realizado o mapeamento já se sabe onde inserir as pontas de prova. Obviamente, é necessária a inserção de comandos para a definição das pontas de prova, declarações de variáveis auxiliares e comandos para a manipulação de arquivos; todos esses comandos são absolutamente vinculados à linguagem de programação da unidade em teste. Porém, onde inserir esses comandos é determinado pela LI, e as modificações a serem feitas consistem somente em alterar os comandos a serem inseridos de uma linguagem para outra.

As demais funções contidas na Figura 2.1 não dependem da linguagem de programação da unidade; o *Cálculo dos Arcos Primitivos* utiliza somente o GFC; a *Geração dos Grafo(i) e Descritores* depende apenas do grafo def; a *Avaliação* utiliza o caminho percorrido pelos casos de teste e os descritores.

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

4.1.2 Aspectos do Projeto Dependentes da Linguagem

A idéia que norteou o projeto/implementação da POKE-TOOL foi tentar isolar os módulos e sub-módulos dependentes da linguagem dos não dependentes, de forma que o usuário configurador necessite intervir somente nesses módulos para obter a nova configuração da POKE-TOOL. Mais ainda, dentro desses módulos, foi feito um esforço para isolar os trechos a serem alterados, de maneira que a intervenção do usuário configurador seja a mais dirigida e localizada possível.

Os módulos **li** e **chanomat** constituem em si uma ferramenta geradora de grafos de fluxo de controle a partir da unidade em LI. O módulo **li** é dependente da linguagem fonte, pois é quem faz a tradução da unidade em teste para a unidade em teste em LI. Para obter a unidade em LI é necessário que o usuário configurador atue no módulo **li** para que o mesmo faça a tradução. Essa tarefa envolve a definição de tabelas e a criação de rotinas específicas que realizem a tradução; essas rotinas têm uma forma fixa e, em geral, manipulam estruturas de dados simples [CAR91]. O módulo **chanomat** tem como entrada a unidade em LI e,

portanto, não necessita ser alterado.

O módulo da POKE-TOOL que faz a maior parte da análise estática do código fonte é o módulo pokernel. Dentro do pokernel, o responsável pela análise estática é o sub-módulo parserli. Este sub-módulo realiza duas funções extremamente dependentes da linguagem de programação que são a *Extensão do Grafo de Fluxo de Controle* e a *Instrumentação*. Como discutido na Seção 3.6.2, essas duas funções são realizadas concomitantemente através de um analisador sintático da LI; esse analisador vai ativar um analisador *específico* para determinar as definições nas declarações, comandos e condições da unidade em teste apontadas pela unidade em LI, e identificar os pontos de introdução das pontas de prova e os comandos adicionais necessários para a geração da versão *instrumentada* da unidade em teste.

O analisador sintático da LI é genérico para qualquer configuração da POKE-TOOL, pois a ferramenta necessita apenas da unidade escrita em LI para gerar o GFC; portanto, sempre será preciso ter uma versão da unidade escrita em LI, qualquer que seja a linguagem de programação. O analisador específico faz a análise do código fonte da unidade em teste propriamente dito. Este analisador específico da linguagem fonte é obtido através de dois analisadores genéricos, um léxico e um sintático, que a POKE-TOOL possui. Para instanciá-los para uma dada linguagem é necessário prover tabelas e rotinas específicas para os dois analisadores. A tarefa de prover as tabelas e as rotinas é do usuário configurador da POKE-TOOL; as rotinas fornecidas devem acessar uma série de estruturas de dados já fornecidas pela ferramenta (por exemplo, o vetor info_no) e utilizar algumas funções e procedimentos também fornecidos (por exemplo, os procedimentos que manipulam "bits"). Ainda, para finalmente obter o analisador específico da POKE-TOOL para uma dada linguagem é necessário compilar e ligar essas novas rotinas com os demais arquivos da POKE-TOOL.

A *Instrumentação* consiste na ativação de procedimentos que *escrevem* o código adicional necessário na nova versão da unidade em teste. Os pontos onde inserir esse código são determinados em função dos comandos da LI e identificados pelo analisador sintático da LI, que ativa os procedimentos de inserção de código. Como a LI é fixa, os pontos onde ativar a inserção de código são fixos; logo, não é necessário alterar o analisador da LI para configurar a POKE-TOOL para instrumentar unidades em uma nova linguagem. Porém, é necessário alterar os procedimentos que inserem o código adicional na versão instrumentada, pois este código varia de linguagem para linguagem.

4.1.3 Passos para a Configuração de uma nova Linguagem

Nesta seção fazemos uma relação das tarefas a serem realizadas por um usuário configurador da POKE-TOOL para obter uma configuração da ferramenta para

uma nova linguagem. As tarefas abaixo descritas estão detalhadas em [CHA91a].

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.4 Dificuldades encontradas na Configuração para a Linguagem C

A linguagem C possui características muito marcantes de ALGOL68 que não foram herdadas pelas demais linguagens "algol-like's" como Pascal, Modula-2, etc.

Entre essas características, encontra-se uma estrutura de blocos que permite a declaração de variáveis em todo início de bloco e comandos, expressões, e blocos de comandos que sempre retornam um valor. As implicações da primeira característica foram discutidas na Seção 3.6.2 quando apresentamos a implementação realizada para a *Extensão do Grafo de Fluxo de Controle* no sub-módulo **parserli**. A segunda característica tem implicação também na *Extensão do Grafo de Fluxo de Controle*. O fato de uma expressão de atribuição poder retornar um valor implica que essa expressão pode fazer parte de uma condição em C, o que torna possível ocorrer uma definição de variável em uma condição. Dessa maneira, é preciso analisarmos também as condições para a determinação do grafo def.

Dessa maneira, para a linguagem C, foram necessários três tipos de "analisadores" específicos: um para as declarações ($DCL), um para os comandos ($S) e um para condições ($C). Na verdade, esses três analisadores são o mesmo analisador sintático genérico da POKE-TOOL, porém configurados diferentemente para cada uma dessas situações. O que diferencia as configurações é o *grafo sintático inicial* (ou produção inicial) que o analisador genérico utiliza para analisar uma declaração, um comando ou uma condição. Essa é uma característica muito interessante dessa solução genérica de análise do código fonte adotada, pois a *unidade em LI* possui apontadores para trechos específicos do código fonte da unidade em teste, mas não para todo código fonte da unidade, como exigiria um analisador sintático descendente (mesmo genérico) que analisasse declarações, comandos ou condições. Assim, com o analisador genérico dirigido por grafos sintáticos, é possível iniciar a análise através do grafo sintático que começa a descrição das *declarações*, ou dos *comandos*, ou das *condições*, sendo necessário acessar somente o código relativo a essas construções da linguagem.

O mapeamento da linguagem fonte para a LI e da LI para a linguagem fonte pode não ser perfeito. Isto pode ocorrer e dificultar as demais tarefas da POKE-TOOL. No caso da linguagem C, tivemos esses problemas.

Um problema de mapeamento que foi detectado está relacionado com os sinônimos que a linguagem C permite dar às especificações das variáveis. Por exemplo, o comando

int x;

está declarando uma variável de nome *x* de tipo *inteiro*, porém, é permitido dar um sinônimo à especificação int da variável *x*, que poderia ser INT. O mapeamento C para LI, realizado pela presente configuração do módulo li, não detecta que INT é um sinônimo, pois isto implicaria ter de analisar todos os arquivos que estão sendo incluído ("include files") na unidade em teste. Dessa maneira, quando estamos dentro do corpo da unidade em teste e temos a declaração

INT x;

a tradução obtida é

$S1 100 6 10.

Esta situação causou muitos problemas, pois poderíamos chamar o analisador de comandos para uma declaração; para resolver esta situação adotamos a seguinte solução: quando o analisador da LI encontra um comando ($S), ele chama um procedimento que chama o analisador específico de declarações; se o analisador de declarações obtém sucesso é porque temos uma declaração através de sinônimos, e o procedimento retorna para o analisador da LI; caso contrário, trata-se de um comando e este procedimento chama o analisador específico de comandos.

Outro problema relacionado com o mapeamento da LI para C (ou vice-versa) é que alguns comandos da LI não apontam para todo o código fonte relativo ao comando traduzido, somente para o essencial. É o caso, por exemplo, do comando

```
$FOR    100    3    5
$S1     104    4    5
$C1     109    6    5
$S2     116    3    5
$S3     120    1    5
```

obtido do comando "**for**(i=0; i < 5; i++);" da linguagem C; o átomo $FOR da LI aponta para a cadeia "for", $S1 para "i=0;", $C1 para "i < 5;", $S2 para "i++" e $S3 para ";". Note-se que os parênteses do comando "for" foram "perdidos" com os ponteiros; logo, é necessário que, durante a geração do comando "for" da nova unidade, seja inserido o restante do comando. Isto implicou na inserção de comandos no analisador da LI (ver Seção 3.6.2), visto que só a cópia do código apontado pelos ponteiros não é suficiente para recuperar o texto da unidade em teste.

4.2 Exemplo de uma Sessão de Trabalho com a POKE-TOOL

Nesta seção apresenta-se um exemplo de utilização da POKE-TOOL para um programa escrito em C. Este programa faz parte do conjunto de programas contido em [KER81]. Esses programas foram originalmente escritos em Pascal; um subconjunto deles foi traduzido manualmente para a linguagem C e testados com o auxílio da POKE-TOOL. No total foram testados 29 programas segundo os três critérios PU implementados; estes programas constituem um "benchmark" para a comparação de critérios estruturais de teste de programas, pois possuem algumas características interessantes: são programas escritos por programadores profissionais, são exemplos de boa técnica de programação e já foram utilizados para avaliar a eficácia de outros critérios de teste [WEY88, WEY90]. Dessa maneira, com o intuito de verificar a eficácia dos critérios PU e sua relação com outros critérios, submeteram-se esses programas à POKE-TOOL; os resultados obtidos são discutidos em [MAL91a].

O programa apresentado aqui é chamado **entab** (pág. 32, [KER81]) e sua função é copiar a entrada via teclado para a saída, substituindo cadeias de espaços em branco por caracteres de tabulação de tal maneira que visualmente a saída é igual à entrada, porém com menor número de caracteres. Este programa encontra-se no arquivo ENTAB.C (ver Apêndice A); por isso, a partir de agora, utilizaremos o nome de seu arquivo para referenciarmos este programa.

Para distinguir entre telas e mensagens do sistema, entradas do usuário e comentários, foi adotada a seguinte convenção: o que for relativo à ferramenta será descrito em "font" de máquina de escrever, o texto entrado pelo usuário é impresso em negrito e os comentários a respeito do funcionamento da ferramenta são escritos em itálico.

A maioria das respostas da ferramenta aparece na forma de arquivos que são apresentados ao usuário através do utilitário *more*. Por isso, iremos referenciar os arquivos contidos no Apêndice A para indicar a maioria das respostas da POKE-TOOL. A referência [CHA91b] contém maiores detalhes de operação da POKE-TOOL.

Para que a sessão de trabalho se inicie é necessário que a POKE-TOOL esteja instalada no diretório onde se encontra o arquivo que contém a unidade em teste. O arquivo ENTAB.C encontra-se, junto com a POKE-TOOL, no diretório `C:\USR\CHAIM\POKETOOL`.

Para iniciar a sessão de trabalho executa-se o programa `poketool` *com o parâmetro* `-i`.

```
C:\USR\CHAIM\POKETOOL>poketool -i

POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2
```

```
Bem-vindo !!

                              INICIACAO

Voce deseja iniciar uma nova sessao de trabalho (S/N) [S] ?
==>s

Mensagens-------------------------------------------------------------
```

A primeira tela é a INICIAÇÃO; pergunta-se ao usuário se deseja iniciar uma nova sessão de trabalho; se o usuário teclar qualquer tecla diferente de 'N' ou 'n' a POKE-TOOL inicia a sessão desde o princípio a partir da fase estática. Caso 'N' ou 'n' seja teclado, a POKE-TOOL solicita ao usuário o diretório onde se encontram os dados da sessão de trabalho a ser recuperada. O símbolo que se encontra entre colchetes é considerado "default", ou seja, qualquer tecla, a menos daquelas que selecionam a outra opção, provocam a sua seleção.

Neste exemplo, iremos iniciar uma sessão de teste com a POKE-TOOL desde o princípio.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

 Bem-vindo !!

                              INICIACAO

Posso apagar os arquivos que estao nesse diretorio (S/N) [N] ?
==>s

Mensagens-------------------------------------------------------------

* * Apagando arquivos da ultima sessao de trabalho * *
```

Antes de iniciar a sessão propriamente dita, ainda na tela INICIAÇÃO, a POKE-TOOL pergunta ao usuário se ele deseja apagar os arquivos gerados pela POKE-TOOL no diretório de trabalho. O procedimento correto é sempre apagar os arquivos antigos que estejam no diretório de trabalho, pois pressupõe-se que essas informações já tenham sido salvas em um sub-diretório.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Vers.0.2

 Bem-vindo !!

                              INICIACAO


Entre com o nome do arquivo que contem a unidade a ser testada (digite
"fim" para terminar a sessao):

==>entab.c


Mensagens-------------------------------------------------------------

* * Determinando o Grafo de Fluxo de Controle * *

* * Ferramenta Geradora de GFC's foi bem sucedida * *

* * Calculando os arcos primitivos ... * *

* * Carregando Tabela de Transicao lexica ... * *

* * Fazendo a analise sintatica do codigo fonte ... * *

* * Gerando descritores ... * *

* * O Nucleo da POKE-TOOL foi bem sucedido * *
```

Em seguida, o usuário deve entrar com o nome do arquivo que contém a unidade em teste. Note-se que, na presente configuração da POKE-TOOL, este arquivo deve conter unicamente a unidade em teste.

Quando o usuário digita a tecla "ENTER", a POKE-TOOL dá início à fase estática da sessão de trabalho que consiste na análise do código fonte através dos módulos **li**, **chanomat** *e* **pokernel**. *No decorrer desta fase, a ferramenta envia uma série de mensagens que indicam o andamento do processo de análise do código fonte.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                        MENU PRINCIPAL

a. Visualizacao Arquivos
b. Gera Programa Executavel
c. Executa Caso de Teste
d. Avaliacao Caso de Teste
e. Termina Sessao

Entre com a opcao desejada:

==>a


Mensagens-------------------------------------------------------------
```

Terminado o processo de preparação da unidade em teste é apresentada a tela MENU PRINCIPAL, indicando que a sessão de trabalho pode prosseguir. Neste ponto a POKE-TOOL já criou os arquivos ENTAB.LI, ENTAB.NLI, ENTAB.GFC, ARCPRIM.TES, GRAFODEF.TES, TESTEPROG.C, PUASSOC.TES, PDUPATHS.TES, DES_PU.TES, DES_PUDU.TES e DES_PDU.TES.

Selecionando a opção **a** *no MENU PRINCIPAL, o usuário poderá visualizar os resultados obtidos pela POKE-TOOL através da tela VISUALIZAÇÃO DE ARQUIVOS. Esta tela apresenta tanto resultados da fase estática como da fase dinâmica.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                      VISUALIZACAO DE ARQUIVOS

a. Arcos Primitivos                          i. Caminhos percorridos pelos Casos
b. Grafo Def                                    de Teste
c. Arquivo Modificado                        j. Associacoes executadas para o
d. Potenciais Du-caminhos Requeridos            criterio todos-potenciais-usos
```

```
e. Associacoes Requeridas                k. Associacoes executadas para o
f. Entrada dos Casos de teste               criterio todos-potenciais-usos/du
g. Entrada do teclado                    l. Potenciais-du-caminhos executados
h. Saida dos Casos de Teste              m. Retorna para o Menu Principal

Entre com a opcao desejada:

==>a

Mensagens-------------------------------------------------------------
```

O usuário, ao selecionar alguma das opções do menu acima, provoca uma chamada de sistema para o utilitário "more"; é apresentado na tela o arquivo associado à opção selecionada. Em seguida a tela VISUALIZAÇÃO DE ARQUIVOS é apresentada novamente.

As opções a, b, c, d e e *do menu apresentam os resultados da análise estática da unidade e estão associadas, respectivamente, aos arquivos ARCPRIM.TES, GRAFODEF.TES, TESTEPROG.TES, PDUPATHS.TES e PUASSOC.TES. Na tela acima é selecionada a opção* a*; é então mostrado o arquivo ARCPRIM.TES que se encontra no Apêndice A.*

As demais opções estão associadas à fase dinâmica da ferramenta e são arquivos gerados pela execução e avaliação dos casos de teste. No atual ponto da sessão de trabalho nenhum desses arquivos foi gerado ainda, pois nenhum caso de teste foi executado nem avaliado. Se o usuário tentar selecionar alguma dessas opções a ferramenta enviará mensagem indicando esse fato.

Para executar e avaliar casos de teste é necessário retornar à tela MENU PRINCIPAL.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                              MENU PRINCIPAL

a. Visualizacao Arquivos
b. Gera Programa Executavel
c. Executa Caso de Teste
d. Avaliacao Caso de Teste
e. Termina Sessao

Entre com a opcao desejada:
```

```
==>b

Mensagens----------------------------------------------------------
```

Antes de executar um caso de teste é necessário gerar o programa executável que contém a unidade em teste instrumentada. Este programa será executado quando forem submetidos os casos de teste. Para gerar este programa deve-se selecionar a opção **b** *do MENU PRINCIPAL.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                        GERA PROGRAMA EXECUTAVEL


OBSERVACAO: Para gerar o programa executavel para teste, voce recebe
            o "prompt" do sistema operacional. Voce devera' compilar
            e "linkar" o arquivo TESTEPROG.C junto com os seus
            outros arquivos no lugar do modulo a ser testado. O novo
            programa executavel devera' ter o nome TESTEPROG.

Digite qualquer tecla para entrar no "shell" do sistema operacional...

Para retornar 'a POKE-TOOL digite "exit"...
```

A tela GERA PROGRAMA EXECUTÁVEL apresenta uma mensagem indicando como o usuário deve proceder para gerar o programa executável. Em seguida, o usuário recebe o "prompt" do sistema operacional para compilar a unidade em teste instrumentada e ligá-la com as demais unidades.

```
C:\USR\CHAIM\POKETOOL>make testeprog
```

Neste caso foi editado um arquivo "makefile" que compila e liga as unidades que compõem o programa executável.

```
testeprog:
tcc -c testeprog.c -I\usr\bin\tc\lib -I\usr\bin\tc\include
tcc -c uni_int.c -I\usr\bin\tc\lib -I\usr\bin\tc\include
        tlink testeprog.obj uni_int.obj testeprog
```

Note-se que o arquivo UNI_INT.OBJ é ligado com o arquivo TESTEPROG.OBJ para obter o programa TESTEPROG.EXE. Isto ocorre porque o arquivo UNI_INT.C contém a função (main) que chama a unidade contida em TESTEPROG.C.

```
C:\USR\CHAIM\POKETOOL>exit
```

Ao retornar à POKE-TOOL retorna-se à tela MENU PRINCIPAL.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                              MENU PRINCIPAL

a. Visualizacao Arquivos
b. Gera Programa Executavel
c. Executa Caso de Teste
d. Avaliacao Caso de Teste
e. Termina Sessao

Entre com a opcao desejada:

==>c


Mensagens-------------------------------------------------------------
```

Para executar um caso de teste basta selecionar a opção c *e a tela EXECUTA CASO DE TESTE será apresentada.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                          EXECUTA CASO DE TESTE

Seu programa necessita de parametros de entrada (S/N) [N]?

==>n

Seu programa tem entrada pelo teclado (S/N) [N]?

==>s
```

```
Mensagens-------------------------------------------------------------

* * A saida esta' sendo direcionada para o arquivo output.tes * *
```

Na primeira vez que se executa um caso de teste a tela EXECUTA CASO DE TESTE pergunta se o programa aceita parâmetros de entrada na linha de comandos e se o programa aceita entrada via teclado. O usuário responde essas perguntas e o primeiro caso de teste é executado. Nas próximas execuções de casos de teste essas perguntas não serão feitas.

```
col 1   2   34  rest

col 1 2 34 rest
Tecle qualquer tecla para retornar 'a POKE-TOOL ...
```

A unidade ENTAB.C não aceita parâmetros de entrada através da linha de comandos mas, se aceitasse, estes parâmetros de entrada seriam salvos no arquivo INPUT1.TES. A entrada via teclado, a saída na tela do caso de teste e o caminho percorrido pelos casos de teste são salvos, respectivamente, nos arquivos TEC1.TES, OUTPUT1.TES e PATH1.TES. O número indica que são arquivos associados ao primeiro caso de teste. As demais entradas via teclado, saídas na tela e caminhos percorridos para os outros casos de teste executados para o programa ENTAB.C são apresentados no Apêndice A. Depois de executado um caso de teste a tela MENU PRINCIPAL é reapresentada e o usuário pode decidir entre "rodar" mais um caso de teste ou avaliar os já executados com relação a algum critério PU.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                              MENU PRINCIPAL

a. Visualizacao Arquivos
b. Gera Programa Executavel
c. Executa Caso de Teste
d. Avaliacao Caso de Teste
e. Termina Sessao

Entre com a opcao desejada:

==>d
```

Mensagens----------

Selecionando-se a opção **d**, *obtém-se a tela AVALIAÇÃO.*

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                        AVALIACAO

a. Criterio Todos-Potenciais-Usos

b. Criterio Todos-Potenciais-Usos/du

c. Criterio Todos-Potenciais-Du-Caminhos

d. Retorna para Menu Principal

Entre com a opcao desejada:

==>a
```

Mensagens----------

```
* * Realizando a avaliacao do caso de teste * *

* * Avaliacao do caso de teste foi bem sucedida * *

* * Terminada a avaliacao de um caso de teste * *
```

O usuário tem três critérios PU implementados na POKE-TOOL que ele pode selecionar através do menu da tela AVALIAÇÃO. O usuário, ao selecionar um dos critérios, por exemplo todos-potenciais-usos, faz com que a POKE-TOOL dê início ao processo de avaliação; durante a avaliação a ferramenta envia mensagens que indicam o andamento do processo. Terminada a avaliação, a POKE-TOOL apresenta o arquivo PUOUTPUT.TES que contém as associações não executadas pelo conjunto de casos de teste avaliado. Depois de apresentado o arquivo, a tela AVALIAÇÃO é reapresentada. Se o usuário selecionar os critérios todos-potenciais-usos e todos-potenciais-du-caminhos para avaliação do conjunto de casos de teste, os arquivos PUDUOUTPUT.TES e PDUOUTPUT.TES são, res-

pectivamente, apresentados ao final. No Apêndice A temos os arquivos PUOUTPUT.TES, PUDUOUTPUT.TES e PDUOUTPUT.TES obtidos após a avaliação de oito casos de teste do exemplo ENTAB.C.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                              AVALIACAO

a. Criterio Todos-Potenciais-Usos

b. Criterio Todos-Potenciais-Usos/du

c. Criterio Todos-Potenciais-Du-Caminhos

d. Retorna para Menu Principal

Entre com a opcao desejada:

==>d

Mensagens-------------------------------------------------------------


POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                              MENU PRINCIPAL

a. Visualizacao Arquivos
b. Gera Programa Executavel
c. Executa Caso de Teste
d. Avaliacao Caso de Teste
e. Termina Sessao

Entre com a opcao desejada:

==>a


Mensagens-------------------------------------------------------------
```

Neste ponto já é possível observar os resultados obtidos da fase dinâmica da sessão de trabalho.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                    VISUALIZACAO DE ARQUIVOS

a. Arcos Primitivos                          i. Caminhos percorridos pelos Casos
b. Grafo Def                                    de Teste
c. Arquivo Modificado                        j. Associacoes executadas para o
d. Potenciais Du-caminhos Requeridos            criterio todos-potenciais-usos
e. Associacoes Requeridas                    k. Associacoes executadas para o
f. Entrada dos Casos de teste                   criterio todos-potenciais-usos/du
g. Entrada do teclado                        l. Potenciais-du-caminhos executados
h. Saida dos Casos de Teste                  m. Retorna para o Menu Principal

Entre com a opcao desejada:
==>g
Entre com o numero do caso de teste (ultimo caso de teste numero 8):
==>7
Mensagens-------------------------------------------------------------
```

As opções **f**, **g**, **h** *e* **i** *apresentam, respectivamente, os pârametros de entrada, entradas via teclado, as saídas na tela e os caminhos percorridos pelos casos de teste. Porém, essas opções estão relacionadas com vários arquivos porque existe um arquivo para cada caso de teste. Quando o usuário seleciona, por exemplo, a entrada do teclado, a POKE-TOOL pergunta a qual caso de teste o usuário está se referindo; se ele selecionar o sétimo caso de teste, a POKE-TOOL irá apresentar o arquivo TEC7.TES. Fato análogo ocorre com as outras opções acima e os conjuntos de arquivos associados.*

As opções **j**, **k**, **l** *apresentam as associações e caminhos efetivamente exercitados pelos casos de teste avaliados. Os arquivos associados com essas opções são, respectivamente, EXEC_PU.TES, EXEC_PUDU.TES e EXEC_PDU.TES. Essas opções apresentam os resultados da última avaliação efetuada; pode ocorrer de terem sido executados casos de teste adicionais que não foram avaliados; portanto, para apresentar o resultado atualizado deve-se primeiramente avaliar os casos de teste. O Apêndice A contém os arquivos acima obtidos para o exemplo ENTAB.C depois da avaliação de oito casos de teste.*

Para retornar ao MENU PRINCIPAL, deve-se selecionar a opção **m**.

```
POKETOOL - Ferramenta de Apoio aos Criterios Potenciais Usos - Ver.0.2

                        MENU PRINCIPAL

a. Visualizacao Arquivos
b. Gera Programa Executavel
c. Executa Caso de Teste
d. Avaliacao Caso de Teste
e. Termina Sessao

Entre com a opcao desejada:

==>e

Salva sessao de trabalho em um diretorio (S/N) [S]?

==>s


Mensagens-------------------------------------------------------------

* * Salvando base de dados de teste no diretorio \USR\CHAIM\POKETOOL\
    entab * *
```

Selecionando-se a opção **e** *é iniciado o processo de término da sessão. A POKE-TOOL pergunta ao usuário se ele deseja que os arquivos gerados pela ferramenta sejam salvos em um sub-diretório com o nome da unidade; em caso afirmativo, a POKE-TOOL cria, se necessário, este sub-diretório e salva os arquivos nele; em caso contrário, a POKE-TOOL termina a execução sem salvar as informações geradas.*

Capítulo 5

Conclusões

5.1 POKE-TOOL – Uma Ferramenta para Suporte ao Teste Estrutural de Programas

Foi desenvolvida uma ferramenta - POKE-TOOL - para suporte à aplicação de alguns critérios da família de Critérios Potenciais Usos (PU), a saber, *todos-potenciais-usos*, *todos-potenciais-usos/du* e *todos-potenciais-du-caminhos*. A POKE-TOOL provê a funcionalidade básica requerida por uma ferramenta de teste estrutural de programas que consiste, segundo Deutsch [DEU82], em:

(1) analisar o código fonte e criar uma base de dados;

(2) gerar relatórios baseados em análise estática do código fonte;

(3) instrumentar o código fonte;

(4) analisar os resultados e gerar relatórios; e

(5) gerar relatórios para auxiliar a organização das atividades de teste e derivar conjuntos de casos de testes específicos.

A POKE-TOOL analisa o código fonte obtendo o grafo de fluxo de controle e determinando os caminhos e associações requeridos pelos critérios PU; esses caminhos e associações são listados em relatórios que podem ajudar o usuário a projetar seus casos de teste; a POKE-TOOL gera também uma nova versão - versão instrumentada - da unidade em teste contendo instruções de escrita que fornecem o caminho percorrido pelo caso de teste; os casos de teste são avaliados e a ferramenta fornece relatórios que indicam quais os caminhos (associações) que não foram ainda satisfeitos para o critério selecionado, bem como os caminhos (associações) que foram satisfeitos para esse critério; ainda, a POKE-TOOL fornece uma

série de informações (entradas, saídas e caminhos executados pelo casos de teste) que retratam como foi realizada e organizada a atividade de teste. Observando-se as atividades suportadas pela POKE-TOOL, nota-se que elas implementam, quase na sua totalidade, as atividades discriminadas por Deutsch como necessárias para a caracterização de uma ferramenta de teste. Dessa maneira, pode se concluir que a POKE-TOOL é uma *ferramenta de teste estrutural de programas.*

A POKE-TOOL tem uma característica que a distingue das demais ferramentas de teste estrutural de software: ela é uma ferramenta *configurável*, ou seja, é possível obter instanciações da ferramenta para que ela suporte outras linguagens de programação.

A atual configuração da POKE-TOOL suporta o teste de programas escritos na linguagem C. Esta configuração foi validada através da realização de um "benchmark" dos critérios PU utilizando um conjunto de programas da literatura [KER81]. Este "benchmark" constituiu-se no teste de 29 programas segundo os três critérios PU implementados na POKE-TOOL. Os resultados obtidos desse "benchmark" [MAL91a] são promissores e indicam que os critérios são factíveis de serem utilizados em ambiente industrial. Mais ainda, a utilização da POKE-TOOL nesse trabalho mostrou que a ferramenta implementada é uma versão inicial de um produto e não apenas um protótipo.

O projeto/implementação da POKE-TOOL, ou melhor, a automação dos critérios Potenciais Usos, envolveu a implementação de algoritmos encontrados na literatura e o desenvolvimento de novos algoritmos.

A POKE-TOOL utiliza o conceito de arcos primitivos no seu mecanismo de avaliação de casos de teste; este fato implicou na necessidade de implementação do algoritmo de cálculo dos arcos primitivos de fluxo de dados (REHFLUXDA), que é uma variação do algoritmo de Chusho. Como pode-se ver da descrição apresentada desse algoritmo, existe uma distância grande entre a descrição em alto nível e sua implementação; alguns dos conceitos contidos no algoritmo exigiram soluções sofisticadas em termos de implementação e essas soluções são originais.

Para a determinação dos grafo(i) e descritores foi desenvolvido e implementado um algoritmo otimizado que gera os *grafo(i)*, utilizados na determinação das *associações*, *caminhos* e *descritores* dos três critérios PU implementados. Foi estudada outra abordagem [CHA89] para determinação dos potenciais-du-caminhos requeridos pelo critério todos-potenciais-du-caminhos mas verificou-se que essa abordagem, apesar de formalizada matematicamente, possui custo computacional igual, é mais difícil de programar e não é tão otimizada, quando comparada com a utilização dos grafo(i).

Outro algoritmo que foi desenvolvido é o *algoritmo de Avaliação.* Este algoritmo simula as transições realizadas por um conjunto de autômatos finitos, obtidos dos descritores do critério selecionado, tendo como cadeia de entrada o caminho executado. O algoritmo realiza as transições "de uma vez", isto é, ele dá uma

única passada pelo arquivo que contém o caminho percorrido e, à medida que ele vai lendo os nós percorridos, realiza as transições nos autômatos; assim, depois de ler o último nó percorrido, todos os autômatos (descritores) estão avaliados. Deve-se observar que a utilização de descritores para representar caminhos e associações requeridos por um critério [MAL91c] provê uma uniformidade no mecanismo de avaliação; o algoritmo necessita apenas dos descritores, do caminho executado e do critério selecionado para avaliar um caso de teste; outras implementações [FRA85] de critérios semelhantes utilizam uma abordagem (casamento de padrão) para avaliar um critério que exige caminhos e outra abordagem (autômatos finitos) para avaliar um critério que exige associações.

Assim, com o desenvolvimento desses algoritmos mais os algoritmos de análise léxica e sintática genéricas responsáveis pela análise do código fonte, temos o cerne da POKE-TOOL — a ferramenta de suporte à aplicação dos critérios Potenciais Usos — implementado e, com a realização do "benchmark", temos a validação preliminar desse cerne.

5.2 Trabalhos Futuros

A atual interface implementada na POKE-TOOL é simples e necessita acrescentar recursos gráficos que permitam apresentar ao usuário o grafo de fluxo de controle da unidade em teste e tornem a ferramenta mais amigável. A simplicidade da interface se deve à ênfase dada nesse trabalho à implementação e validação do cerne operacional da ferramenta; esses dois trabalhos já estão em andamento e essas facilidades serão incorporadas na próxima versão da POKE-TOOL.

A abordagem utilizada para avaliar os critérios Potenciais Usos através de descritores proveu uma uniformidade não só aplicável a esses critérios, mas à maioria dos critérios estruturais de teste baseados na Análise de Fluxos de Dados. Dessa maneira, uma tarefa interessante é incorporar outros critérios de teste na ferramenta utilizando o conceito de descritores. Isto viabilizará estudos que permitirão avaliar as classes de defeitos detectados e não detectados por diversos critérios.

Uma característica inerente aos critérios estruturais de teste de programa é que, invariavelmente, são selecionados caminhos de teste não executáveis, ou seja, não existem valores possíveis para as variáveis de entrada que causem a execução desse particular caminho. Os critérios Potenciais Usos, como critérios estruturais, não são diferentes e geram também caminhos e associações não executáveis; nos 29 programas testados com a POKE-TOOL, somente 3 programas não apresentaram caminhos desse tipo [MAL91a, MAL91b]. Portanto, uma necessidade detectada é a incorporação de mecanismos que auxiliem o usuário a determinar se um caminho é não executável.

A POKE-TOOL é uma ferramenta que foi projetada para ser configurável para várias linguagens. Atualmente, a POKE-TOOL está configurada para C e o

módulo **li** já foi configurado para C e COBOL [CAR91]. Atualmente, mais duas configurações da POKE-TOOL estão sendo desenvolvidas: para COBOL e para FORTRAN.

Para melhorar a operacionalidade da POKE-TOOL é necessário incluir algumas facilidades adicionais. Uma dessas facilidades seriam mecanismos para a manutenção da sessão de trabalho; esta manutenção consistiria em, por exemplo, prover recursos para retirar um caso de teste da sessão de trabalho. Outras necessidades operacionais da POKE-TOOL são: meios para que o usuário submeta casos de teste em "batch"; um mecanismo de compilação e ligação automática da unidade em teste instrumentada e os arquivos que compõem o programa executável; e geração automática de programas "drivers" e "stubs" eventualmente necessários para realização do teste da unidade.

A presente forma da POKE-TOOL é o núcleo de um ambiente que deverá evoluir no sentido de incorporar mais recursos para gerenciar e efetuar as várias atividades envolvidas no teste de software.

Apêndice A

Um Exemplo Completo — ENTAB.C

Neste apêndice é apresentado um exemplo completo — ENTAB.C, extraído do "benchmark"—, utilizado para ilustrar as informações estáticas e dinâmicas geradas pela ferramenta POKE-TOOL.

A.1 Informações Estáticas

Nesta seção são organizados as informações estáticas geradas pela POKE-TOOL, para o programa ENTAB.C; estas informações estão contidas em arquivos gerados pela POKE-TOOL, durante a fase de análise estática e são reproduzidas a seguir, sem qualquer alteração.

Arquivo Entab.c

```
#include "cte.c"

void entab()

/*  Substitui strings de brancos por tabs. Produz visualmente a
    mesma saida, mas com menos caracteres                          */


{

 int c,col,newcol;
 int tabstops[MAXLINE];
```

```
settabs(tabstops);
col = 0;

do
{
  newcol = col;
  while ((c = getchar()) == BLANK)
  {
    newcol++;
    if (tabpos(newcol,tabstops))
      {
        putchar(TAB);
        col = newcol;
      }
  }
  while (col<newcol)
  {
   putchar(BLANK);
   col++;
  }
  if (c!=ENDFILE)
  {
    putchar(c);
    if (c==NEWLINE)
      col = 0;
    else
      col++;
  }
 }while (c!=ENDFILE);
}
```

Arquivo Entab.li

```
$DCL                    1      182       1
{                     182        1      13
$DCL                  189       17      15
$DCL                  209       22      16
$S1                   234       18      17
$S2                   255        8      18
$REPEAT               268        2      20
{                     273        1      21
$S3                   279       13      22
```

$WHILE	297	5	23
$C(01)1	303	26	23
{	334	1	24
$S4	342	9	25
$IF	358	2	26
$C(01)2	361	25	26
{	395	1	27
$S5	407	13	28
$S6	431	13	29
}	453	1	30
}	459	1	31
$WHILE	465	5	32
$C(01)3	471	12	32
{	488	1	33
$S7	495	15	34
$S8	516	6	35
}	527	1	36
$IF	533	2	37
$C(01)4	536	12	37
{	553	1	38
$S9	561	11	39
$IF	579	2	40
$C(01)5	582	12	40
$S10	603	8	41
$ELSE	618	4	42
$S11	631	6	43
}	642	1	44
}	646	1	45
$UNTIL	647	5	45
$NC(01)6	653	12	45
}	668	1	46

Arquivo Entab.nli

$DCL	1	1	181	1
{	1	182	1	13
$DCL	1	189	17	15
$DCL	1	209	22	16
$S1	1	234	18	17
$S2	1	255	8	18
$REPEAT	2	268	2	20

```
{                2    273     1    21
$S3              2    279    13    22
$WHILE           3    297     5    23
$C(01)1          3    303    26    23
{                4    334     1    24
$S4              4    342     9    25
$IF              4    358     2    26
$C(01)2          4    361    25    26
{                5    395     1    27
$S5              5    407    13    28
$S6              5    431    13    29
}                5    453     1    30
}                6    459     1    31
$WHILE           7    465     5    32
$C(01)3          7    471    12    32
{                8    488     1    33
$S7              8    495    15    34
$S8              8    516     6    35
}                8    527     1    36
$IF              9    533     2    37
$C(01)4          9    536    12    37
{               10    553     1    38
$S9             10    561    11    39
$IF             10    579     2    40
$C(01)5         10    582    12    40
{               11      0     0     0
$S10            11    603     8    41
}               11      0     0     0
$ELSE           12    618     4    42
{               12      0     0     0
$S11            12    631     6    43
}               12      0     0     0
}               13    642     1    44
}               14    646     1    45
$UNTIL          14    647     5    45
$NC(01)6        14    653    12    45
}               15    668     0    46
```

Arquivo Entab.gfc

```
15
```

```
1
   2 0
2
   3 0
3
   4 7 0
4
   5 6 0
5
   6 0
6
   3 0
7
   8 9 0
8
   7 0
9
   10 14 0
10
   11 12 0
11
   13 0
12
   13 0
13
   14 0
14
   2 15 0
15
   0
```

Arquivo arcprim.tes

```
ARCOS PRIMITIVOS DO MODULO entab.c

arco ( 4, 5) e' primitivo
arco ( 4, 6) e' primitivo
arco ( 7, 8) e' primitivo
arco ( 9,14) e' primitivo
arco (10,11) e' primitivo
arco (10,12) e' primitivo
arco (14, 2) e' primitivo
```

```
arco (14,15) e' primitivo
```

Arquivo Testeprog.c

```
#define ponta_de_prova(num) if(printed_nodes % 10) {++printed_nodes;
fprintf(path," %d ",num);}\
else {++printed_nodes; fprintf(path," %d\n",num);}

#include <stdio.h>


#include "cte.c"




void entab()

/*  Substitui strings de brancos por tabs. Produz visualmente a
    mesma saida, mas com menos caracteres                          */



/*  1 */        {
                   FILE * path = fopen("path.tes","w");
                   static int printed_nodes = 0;
/*  1 */           int c,col,newcol;
/*  1 */           int tabstops[MAXLINE];
                   ponta_de_prova(1);
/*  1 */           settabs(tabstops);
/*  1 */           col = 0;
/*  2 */           do
/*  2 */           {
                      ponta_de_prova(2);
/*  2 */              newcol = col;
/*  3 */              while((c = getchar()) == BLANK)
/*  4 */              {
                         ponta_de_prova(3);
                         ponta_de_prova(4);
```

```
/*  4 */                newcol++;
/*  4 */                if(tabpos(newcol,tabstops))
/*  5 */                {
                           ponta_de_prova(5);
/*  5 */                   putchar(TAB);
/*  5 */                   col = newcol;
/*  5 */                }
                           ponta_de_prova(6);
/*  6 */             }
                     ponta_de_prova(3);
/*  7 */             while(col<newcol)
/*  8 */             {
                        ponta_de_prova(7);
                        ponta_de_prova(8);
/*  8 */                putchar(BLANK);
/*  8 */                col++;
/*  8 */             }
                     ponta_de_prova(7);
                     ponta_de_prova(9);
/*  9 */             if(c!=ENDFILE)
/* 10 */             {
                        ponta_de_prova(10);
/* 10 */                putchar(c);
/* 10 */                if(c==NEWLINE)
/* 11 */                {
                           ponta_de_prova(11);
/* 11 */                   col = 0;
/* 11 */                }
/* 12 */                else
/* 12 */                {
                           ponta_de_prova(12);
/* 12 */                   col++;
/* 12 */                }
                        ponta_de_prova(13);
/* 13 */             }
                     ponta_de_prova(14);
/* 14 */          }
/* 14 */          while(c!=ENDFILE);
                  ponta_de_prova(15);
                  fclose(path);
/* 15 */       }
```

Arquivo grafodef.tes

```
VARIAVEIS DEFINIDAS nos NOS do modulo entab.c

Variaveis Definidas  = Vars Defs
Variaveis possivelmente definidas por Referencia = Vars Refs


NO'  1
Vars defs:  col
Vars refs:  tabstops
NO'  2
Vars defs:  newcol
Vars refs:
NO'  3
Vars defs:  c
Vars refs:
NO'  4
Vars defs:  newcol
Vars refs:  tabstops
NO'  5
Vars defs:  col
Vars refs:
NO'  6
Vars defs:
Vars refs:
NO'  7
Vars defs:
Vars refs:
NO'  8
Vars defs:  col
Vars refs:
NO'  9
Vars defs:
Vars refs:
NO' 10
Vars defs:
Vars refs:
NO' 11
Vars defs:  col
```

```
Vars refs:
NO' 12
Vars defs:  col
Vars refs:
NO' 13
Vars defs:
Vars refs:
NO' 14
Vars defs:
Vars refs:
NO' 15
Vars defs:
Vars refs:
```

Arquivo puassoc.tes

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


Associacoes requeridas pelo Grafo( 1)

1) <1,(9,14),{ col, tabstops }>
2) <1,(10,12),{ col, tabstops }>
3) <1,(14,15),{ col, tabstops }>
4) <1,(14,15),{ tabstops }>
5) <1,(14,2),{ col, tabstops }>
6) <1,(14,2),{ tabstops }>
7) <1,(10,11),{ col, tabstops }>
8) <1,(8,7),{ tabstops }>
9) <1,(7,8),{ col, tabstops }>
10) <1,(4,6),{ col, tabstops }>
11) <1,(6,3),{ col, tabstops }>
12) <1,(6,3),{ tabstops }>
13) <1,(4,5),{ col, tabstops }>


Associacoes requeridas pelo Grafo( 2)

14) <2,(9,14),{ newcol }>
15) <2,(10,12),{ newcol }>
```

16) <2,(14,15),{ newcol }>
17) <2,(14,2),{ newcol }>
18) <2,(10,11),{ newcol }>
19) <2,(8,7),{ newcol }>
20) <2,(7,8),{ newcol }>
21) <2,(3,4),{ newcol }>

Associacoes requeridas pelo Grafo(3)

22) <3,(9,14),{ c }>
23) <3,(10,12),{ c }>
24) <3,(14,15),{ c }>
25) <3,(2,3),{ c }>
26) <3,(14,2),{ c }>
27) <3,(10,11),{ c }>
28) <3,(8,7),{ c }>
29) <3,(7,8),{ c }>
30) <3,(4,6),{ c }>
31) <3,(6,3),{ c }>
32) <3,(4,5),{ c }>

Associacoes requeridas pelo Grafo(4)

33) <4,(4,6),{ newcol, tabstops }>
34) <4,(9,14),{ newcol, tabstops }>
35) <4,(10,12),{ newcol, tabstops }>
36) <4,(14,15),{ newcol, tabstops }>
37) <4,(2,3),{ tabstops }>
38) <4,(14,2),{ newcol, tabstops }>
39) <4,(10,11),{ newcol, tabstops }>
40) <4,(8,7),{ newcol, tabstops }>
41) <4,(7,8),{ newcol, tabstops }>
42) <4,(3,4),{ newcol, tabstops }>
43) <4,(4,5),{ newcol, tabstops }>

Associacoes requeridas pelo Grafo(5)

44) <5,(14,15),{ col }>

45) <5,(2,3),{ col }>
46) <5,(14,2),{ col }>
47) <5,(9,14),{ col }>
48) <5,(10,12),{ col }>
49) <5,(10,11),{ col }>
50) <5,(7,8),{ col }>
51) <5,(4,6),{ col }>
52) <5,(4,5),{ col }>

Associacoes requeridas pelo Grafo(8)

53) <8,(14,15),{ col }>
54) <8,(3,7),{ col }>
55) <8,(6,3),{ col }>
56) <8,(4,6),{ col }>
57) <8,(4,5),{ col }>
58) <8,(14,2),{ col }>
59) <8,(9,14),{ col }>
60) <8,(10,12),{ col }>
61) <8,(10,11),{ col }>
62) <8,(7,8),{ col }>

Associacoes requeridas pelo Grafo(11)

63) <11,(14,15),{ col }>
64) <11,(9,14),{ col }>
65) <11,(10,12),{ col }>
66) <11,(10,11),{ col }>
67) <11,(7,8),{ col }>
68) <11,(6,3),{ col }>
69) <11,(4,6),{ col }>
70) <11,(4,5),{ col }>
71) <11,(14,2),{ col }>

Associacoes requeridas pelo Grafo(12)

72) <12,(14,15),{ col }>
73) <12,(9,14),{ col }>

```
74) <12,(10,12),{ col }>
75) <12,(10,11),{ col }>
76) <12,(7,8),{ col }>
77) <12,(6,3),{ col }>
78) <12,(4,6),{ col }>
79) <12,(4,5),{ col }>
80) <12,(14,2),{ col }>
```

Arquivo pdupaths.tes

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


Caminhos requeridos pelo Grafo( 1)

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


Caminhos requeridos pelo Grafo( 2)

10) 2 3 7 9 14 15
11) 2 3 7 9 14 2
12) 2 3 7 9 10 12 13 14 15
13) 2 3 7 9 10 12 13 14 2
14) 2 3 7 9 10 11 13 14 15
15) 2 3 7 9 10 11 13 14 2
16) 2 3 7 8 7
17) 2 3 4


Caminhos requeridos pelo Grafo( 3)
```

18) 3 7 9 14 15
19) 3 7 9 14 2 3
20) 3 7 9 10 12 13 14 15
21) 3 7 9 10 12 13 14 2 3
22) 3 7 9 10 11 13 14 15
23) 3 7 9 10 11 13 14 2 3
24) 3 7 8 7
25) 3 4 6 3
26) 3 4 5 6 3

Caminhos requeridos pelo Grafo(4)

27) 4 6 3 7 9 14 15
28) 4 6 3 7 9 14 2 3
29) 4 6 3 7 9 10 12 13 14 15
30) 4 6 3 7 9 10 12 13 14 2 3
31) 4 6 3 7 9 10 11 13 14 15
32) 4 6 3 7 9 10 11 13 14 2 3
33) 4 6 3 7 8 7
34) 4 6 3 4
35) 4 5 6 3 7 9 14 15
36) 4 5 6 3 7 9 14 2 3
37) 4 5 6 3 7 9 10 12 13 14 15
38) 4 5 6 3 7 9 10 12 13 14 2 3
39) 4 5 6 3 7 9 10 11 13 14 15
40) 4 5 6 3 7 9 10 11 13 14 2 3
41) 4 5 6 3 7 8 7
42) 4 5 6 3 4

Caminhos requeridos pelo Grafo(5)

43) 5 6 3 7 9 14 15
44) 5 6 3 7 9 14 2 3
45) 5 6 3 7 9 10 12
46) 5 6 3 7 9 10 11
47) 5 6 3 7 8
48) 5 6 3 4 6
49) 5 6 3 4 5

Caminhos requeridos pelo Grafo(8)

50) 8 7 9 14 15
51) 8 7 9 14 2 3 7
52) 8 7 9 14 2 3 4 6 3
53) 8 7 9 14 2 3 4 5
54) 8 7 9 10 12
55) 8 7 9 10 11
56) 8 7 8

Caminhos requeridos pelo Grafo(11)

57) 11 13 14 15
58) 11 13 14 2 3 7 9 14
59) 11 13 14 2 3 7 9 10 12
60) 11 13 14 2 3 7 9 10 11
61) 11 13 14 2 3 7 8
62) 11 13 14 2 3 4 6 3
63) 11 13 14 2 3 4 5

Caminhos requeridos pelo Grafo(12)

64) 12 13 14 15
65) 12 13 14 2 3 7 9 14
66) 12 13 14 2 3 7 9 10 12
67) 12 13 14 2 3 7 9 10 11
68) 12 13 14 2 3 7 8
69) 12 13 14 2 3 4 6 3
70) 12 13 14 2 3 4 5

Arquivo des_pu.tes

DESCRITORES PARA O CRITERIO TODOS POT-USOS

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

Descritores para o Grafo(1)

```
Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 2 3 7 15

1) N* 1 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 2 3 4 6 7 9 10 13 14 15

2) N* 1 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 2 3 4 6 7 9 10 13 14 15

3) N* 1 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 6 7 9 10 13 14 15

4) N* 1 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

5) N* 1 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 3 4 6 7 9 10 13 14 15

6) N* 1 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

7) N* 1 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 2 3 4 6 7 9 10 13 14 15

8) N* 1 Nnv* 8 [ Nnv* 8 ]* 7
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

9) N* 1 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 2 3 4 6 7 9 10 13 14 15

10) N* 1 Nnv* 4 [ Nnv* 4 ]* 6
Nnv = 2 3 4 6 7 9 10 13 14 15

11) N* 1 Nnv* 6 [ Nnv* 6 ]* 3
Nnv = 2 3 4 6 7 9 10 13 14 15

12) N* 1 Nnv* 6 [ Nnv* 6 ]* 3
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

13) N* 1 Nnv* 4 [ Nnv* 4 ]* 5
```

```
Nnv = 2 3 4 6 7 9 10 13 14 15


Descritores para o Grafo( 2)

Ni = 2 3 4 7 8 9 10 11 12 13 14 15
Nt = 2 4 7 15

14) N* 2 Nnv* 9 [ Nnv* 9 ]* 14
Nnv = 3 7 8 9 10 11 12 13 14 15

15) N* 2 Nnv* 10 [ Nnv* 10 ]* 12
Nnv = 3 7 8 9 10 11 12 13 14 15

16) N* 2 Nnv* 14 [ Nnv* 14 ]* 15
Nnv = 3 7 8 9 10 11 12 13 14 15

17) N* 2 Nnv* 14 [ Nnv* 14 ]* 2
Nnv = 3 7 8 9 10 11 12 13 14 15

18) N* 2 Nnv* 10 [ Nnv* 10 ]* 11
Nnv = 3 7 8 9 10 11 12 13 14 15

19) N* 2 Nnv* 8 [ Nnv* 8 ]* 7
Nnv = 3 7 8 9 10 11 12 13 14 15

20) N* 2 Nnv* 7 [ Nnv* 7 ]* 8
Nnv = 3 7 8 9 10 11 12 13 14 15

21) N* 2 Nnv* 3 [ Nnv* 3 ]* 4
Nnv = 3 7 8 9 10 11 12 13 14 15


Descritores para o Grafo( 3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 7 15

22) N* 3 Nnv* 9 [ Nnv* 9 ]* 14
```

Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

23) N* 3 Nnv* 10 [Nnv* 10]* 12
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

24) N* 3 Nnv* 14 [Nnv* 14]* 15
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

25) N* 3 Nnv* 2 [Nnv* 2]* 3
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

26) N* 3 Nnv* 14 [Nnv* 14]* 2
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

27) N* 3 Nnv* 10 [Nnv* 10]* 11
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

28) N* 3 Nnv* 8 [Nnv* 8]* 7
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

29) N* 3 Nnv* 7 [Nnv* 7]* 8
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

30) N* 3 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

31) N* 3 Nnv* 6 [Nnv* 6]* 3
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

32) N* 3 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14 15

Descritores para o Grafo(4)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 4 7 15

33) N* 4 [Nnv* 4]* 6
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

34) N* 4 Nnv* 9 [Nnv* 9]* 14
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

35) N* 4 Nnv* 10 [Nnv* 10]* 12
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

36) N* 4 Nnv* 14 [Nnv* 14]* 15
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

37) N* 4 Nnv* 2 [Nnv* 2]* 3
Nnv = 2 3 5 6 7 8 9 10 11 12 13 14 15

38) N* 4 Nnv* 14 [Nnv* 14]* 2
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

39) N* 4 Nnv* 10 [Nnv* 10]* 11
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

40) N* 4 Nnv* 8 [Nnv* 8]* 7
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

41) N* 4 Nnv* 7 [Nnv* 7]* 8
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

42) N* 4 Nnv* 3 [Nnv* 3]* 4
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

43) N* 4 [Nnv* 4]* 5
Nnv = 3 5 6 7 8 9 10 11 12 13 14 15

Descritores para o Grafo(5)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 6 8 11 12 15

44) N* 5 Nnv* 14 [Nnv* 14]* 15
Nnv = 2 3 4 6 7 9 10 14 15

45) N* 5 Nnv* 2 [Nnv* 2]* 3
Nnv = 2 3 4 6 7 9 10 14 15

46) N* 5 Nnv* 14 [Nnv* 14]* 2
Nnv = 2 3 4 6 7 9 10 14 15

47) N* 5 Nnv* 9 [Nnv* 9]* 14
Nnv = 2 3 4 6 7 9 10 14 15

48) N* 5 Nnv* 10 [Nnv* 10]* 12
Nnv = 2 3 4 6 7 9 10 14 15

49) N* 5 Nnv* 10 [Nnv* 10]* 11
Nnv = 2 3 4 6 7 9 10 14 15

50) N* 5 Nnv* 7 [Nnv* 7]* 8
Nnv = 2 3 4 6 7 9 10 14 15

51) N* 5 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 3 4 6 7 9 10 14 15

52) N* 5 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 6 7 9 10 14 15

Descritores para o Grafo(8)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 7 8 11 12 15

53) N* 8 Nnv* 14 [Nnv* 14]* 15
Nnv = 2 3 4 6 7 9 10 14 15

54) N* 8 Nnv* 3 [Nnv* 3]* 7
Nnv = 2 3 4 6 7 9 10 14 15

55) N* 8 Nnv* 6 [Nnv* 6]* 3
Nnv = 2 3 4 6 7 9 10 14 15

56) N* 8 Nnv* 4 [Nnv* 4]* 6

Nnv = 2 3 4 6 7 9 10 14 15

57) N* 8 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 6 7 9 10 14 15

58) N* 8 Nnv* 14 [Nnv* 14]* 2
Nnv = 2 3 4 6 7 9 10 14 15

59) N* 8 Nnv* 9 [Nnv* 9]* 14
Nnv = 2 3 4 6 7 9 10 14 15

60) N* 8 Nnv* 10 [Nnv* 10]* 12
Nnv = 2 3 4 6 7 9 10 14 15

61) N* 8 Nnv* 10 [Nnv* 10]* 11
Nnv = 2 3 4 6 7 9 10 14 15

62) N* 8 Nnv* 7 [Nnv* 7]* 8
Nnv = 2 3 4 6 7 9 10 14 15

Descritores para o Grafo(11)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

63) N* 11 Nnv* 14 [Nnv* 14]* 15
Nnv = 2 3 4 6 7 9 10 13 14 15

64) N* 11 Nnv* 9 [Nnv* 9]* 14
Nnv = 2 3 4 6 7 9 10 13 14 15

65) N* 11 Nnv* 10 [Nnv* 10]* 12
Nnv = 2 3 4 6 7 9 10 13 14 15

66) N* 11 Nnv* 10 [Nnv* 10]* 11
Nnv = 2 3 4 6 7 9 10 13 14 15

67) N* 11 Nnv* 7 [Nnv* 7]* 8
Nnv = 2 3 4 6 7 9 10 13 14 15

68) N* 11 Nnv* 6 [Nnv* 6]* 3
Nnv = 2 3 4 6 7 9 10 13 14 15

69) N* 11 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 3 4 6 7 9 10 13 14 15

70) N* 11 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 6 7 9 10 13 14 15

71) N* 11 Nnv* 14 [Nnv* 14]* 2
Nnv = 2 3 4 6 7 9 10 13 14 15

Descritores para o Grafo(12)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

72) N* 12 Nnv* 14 [Nnv* 14]* 15
Nnv = 2 3 4 6 7 9 10 13 14 15

73) N* 12 Nnv* 9 [Nnv* 9]* 14
Nnv = 2 3 4 6 7 9 10 13 14 15

74) N* 12 Nnv* 10 [Nnv* 10]* 12
Nnv = 2 3 4 6 7 9 10 13 14 15

75) N* 12 Nnv* 10 [Nnv* 10]* 11
Nnv = 2 3 4 6 7 9 10 13 14 15

76) N* 12 Nnv* 7 [Nnv* 7]* 8
Nnv = 2 3 4 6 7 9 10 13 14 15

77) N* 12 Nnv* 6 [Nnv* 6]* 3
Nnv = 2 3 4 6 7 9 10 13 14 15

78) N* 12 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 3 4 6 7 9 10 13 14 15

79) N* 12 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 6 7 9 10 13 14 15

80) N* 12 Nnv* 14 [Nnv* 14]* 2
Nnv = 2 3 4 6 7 9 10 13 14 15

Numero Total de Descritores = 80

Arquivo des_pudu

DESCRITORES PARA O CRITERIO TODOS POT-USOS/DU

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

Descritores para o Grafo(1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 2 3 7 15

1) N* 1 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 13 14 15

2) N* 1 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 13 14 15

3) N* 1 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 13 14 15

4) N* 1 Nnvlf* 14 15
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

5) N* 1 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 13 14 15

6) N* 1 Nnvlf* 14 2
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

7) N* 1 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 13 14 15

8) N* 1 Nnvlf* 8 7
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

9) N* 1 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 13 14 15

10) N* 1 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 13 14 15

11) N* 1 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 13 14 15

12) N* 1 Nnvlf* 6 3
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

13) N* 1 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 13 14 15

Descritores para o Grafo(2)

Ni = 2 3 4 7 8 9 10 11 12 13 14 15
Nt = 2 4 7 15

14) N* 2 Nnvlf* 9 14
Nnvlf = 3 7 8 9 10 11 12 13 14 15

15) N* 2 Nnvlf* 10 12
Nnvlf = 3 7 8 9 10 11 12 13 14 15

16) N* 2 Nnvlf* 14 15
Nnvlf = 3 7 8 9 10 11 12 13 14 15

17) N* 2 Nnvlf* 14 2
Nnvlf = 3 7 8 9 10 11 12 13 14 15

18) N* 2 Nnvlf* 10 11
Nnvlf = 3 7 8 9 10 11 12 13 14 15

19) N* 2 Nnvlf* 8 7

Nnvlf = 3 7 8 9 10 11 12 13 14 15

20) N* 2 Nnvlf* 7 8
Nnvlf = 3 7 8 9 10 11 12 13 14 15

21) N* 2 Nnvlf* 3 4
Nnvlf = 3 7 8 9 10 11 12 13 14 15

Descritores para o Grafo(3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 7 15

22) N* 3 Nnvlf* 9 14
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

23) N* 3 Nnvlf* 10 12
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

24) N* 3 Nnvlf* 14 15
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

25) N* 3 Nnvlf* 2 3
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

26) N* 3 Nnvlf* 14 2
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

27) N* 3 Nnvlf* 10 11
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

28) N* 3 Nnvlf* 8 7
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

29) N* 3 Nnvlf* 7 8
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

30) N* 3 Nnvlf* 4 6
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

31) N* 3 Nnvlf* 6 3
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

32) N* 3 Nnvlf* 4 5
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14 15

Descritores para o Grafo(4)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 4 7 15

33) N* 4 6
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

34) N* 4 Nnvlf* 9 14
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

35) N* 4 Nnvlf* 10 12
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

36) N* 4 Nnvlf* 14 15
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

37) N* 4 Nnvlf* 2 3
Nnvlf = 2 3 5 6 7 8 9 10 11 12 13 14 15

38) N* 4 Nnvlf* 14 2
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

39) N* 4 Nnvlf* 10 11
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

40) N* 4 Nnvlf* 8 7
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

41) N* 4 Nnvlf* 7 8
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

42) N* 4 Nnvlf* 3 4
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

43) N* 4 5
Nnvlf = 3 5 6 7 8 9 10 11 12 13 14 15

Descritores para o Grafo(5)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 6 8 11 12 15

44) N* 5 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 14 15

45) N* 5 Nnvlf* 2 3
Nnvlf = 2 3 4 6 7 9 10 14 15

46) N* 5 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 14 15

47) N* 5 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 14 15

48) N* 5 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 14 15

49) N* 5 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 14 15

50) N* 5 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 14 15

51) N* 5 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 14 15

52) N* 5 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 14 15

Descritores para o Grafo(8)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 7 8 11 12 15

53) N* 8 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 14 15

54) N* 8 Nnvlf* 3 7
Nnvlf = 2 3 4 6 7 9 10 14 15

55) N* 8 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 14 15

56) N* 8 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 14 15

57) N* 8 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 14 15

58) N* 8 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 14 15

59) N* 8 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 14 15

60) N* 8 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 14 15

61) N* 8 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 14 15

62) N* 8 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 14 15

Descritores para o Grafo(11)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```
Nt = 3 5 8 11 12 14 15

63) N* 11 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 13 14 15

64) N* 11 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 13 14 15

65) N* 11 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 13 14 15

66) N* 11 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 13 14 15

67) N* 11 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 13 14 15

68) N* 11 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 13 14 15

69) N* 11 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 13 14 15

70) N* 11 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 13 14 15

71) N* 11 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 13 14 15



Descritores para o Grafo(12)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

72) N* 12 Nnvlf* 14 15
Nnvlf = 2 3 4 6 7 9 10 13 14 15

73) N* 12 Nnvlf* 9 14
Nnvlf = 2 3 4 6 7 9 10 13 14 15
```

```
74) N* 12 Nnvlf* 10 12
Nnvlf = 2 3 4 6 7 9 10 13 14 15

75) N* 12 Nnvlf* 10 11
Nnvlf = 2 3 4 6 7 9 10 13 14 15

76) N* 12 Nnvlf* 7 8
Nnvlf = 2 3 4 6 7 9 10 13 14 15

77) N* 12 Nnvlf* 6 3
Nnvlf = 2 3 4 6 7 9 10 13 14 15

78) N* 12 Nnvlf* 4 6
Nnvlf = 2 3 4 6 7 9 10 13 14 15

79) N* 12 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 9 10 13 14 15

80) N* 12 Nnvlf* 14 2
Nnvlf = 2 3 4 6 7 9 10 13 14 15


Numero Total de Descritores = 80
```

Arquivo des_pdu.tes

```
DESCRITORES PARA O CRITERIO TODOS POT-DU-CAMINHOS

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


Descritores para o Grafo( 1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 2 3 7 15

 1) N* 1 Nlf* 9 14 15
 2) N* 1 Nlf* 9 14 2
 3) N* 1 Nlf* 10 12 Nlf* 14 15
 4) N* 1 Nlf* 10 12 Nlf* 14 2
```

5) N* 1 Nlf* 10 11 Nlf* 14 15
6) N* 1 Nlf* 10 11 Nlf* 14 2
7) N* 1 Nlf* 7 8 7
8) N* 1 Nlf* 4 6 3
9) N* 1 Nlf* 4 5 6 3

Descritores para o Grafo(2)

Ni = 2 3 4 7 8 9 10 11 12 13 14 15
Nt = 2 4 7 15

10) N* 2 Nlf* 9 14 15
11) N* 2 Nlf* 9 14 2
12) N* 2 Nlf* 10 12 Nlf* 14 15
13) N* 2 Nlf* 10 12 Nlf* 14 2
14) N* 2 Nlf* 10 11 Nlf* 14 15
15) N* 2 Nlf* 10 11 Nlf* 14 2
16) N* 2 Nlf* 7 8 7
17) N* 2 3 4

Descritores para o Grafo(3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 7 15

18) N* 3 Nlf* 9 14 15
19) N* 3 Nlf* 9 14 2 3
20) N* 3 Nlf* 10 12 Nlf* 14 15
21) N* 3 Nlf* 10 12 Nlf* 14 2 3
22) N* 3 Nlf* 10 11 Nlf* 14 15
23) N* 3 Nlf* 10 11 Nlf* 14 2 3
24) N* 3 7 8 7
25) N* 3 4 6 3
26) N* 3 4 5 6 3

Descritores para o Grafo(4)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Nt = 3 4 7 15

27) N* 4 6 Nlf* 9 14 15
28) N* 4 6 Nlf* 9 14 2 3
29) N* 4 6 Nlf* 10 12 Nlf* 14 15
30) N* 4 6 Nlf* 10 12 Nlf* 14 2 3
31) N* 4 6 Nlf* 10 11 Nlf* 14 15
32) N* 4 6 Nlf* 10 11 Nlf* 14 2 3
33) N* 4 6 Nlf* 7 8 7
34) N* 4 6 3 4
35) N* 4 5 Nlf* 9 14 15
36) N* 4 5 Nlf* 9 14 2 3
37) N* 4 5 Nlf* 10 12 Nlf* 14 15
38) N* 4 5 Nlf* 10 12 Nlf* 14 2 3
39) N* 4 5 Nlf* 10 11 Nlf* 14 15
40) N* 4 5 Nlf* 10 11 Nlf* 14 2 3
41) N* 4 5 Nlf* 7 8 7
42) N* 4 5 Nlf* 3 4

Descritores para o Grafo(5)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 6 8 11 12 15

43) N* 5 Nlf* 9 14 15
44) N* 5 Nlf* 9 14 2 3
45) N* 5 Nlf* 10 12
46) N* 5 Nlf* 10 11
47) N* 5 Nlf* 7 8
48) N* 5 Nlf* 4 6
49) N* 5 Nlf* 4 5

Descritores para o Grafo(8)

Ni = 2 3 4 5 6 7 8 9 10 11 12 14 15
Nt = 3 5 7 8 11 12 15

50) N* 8 Nlf* 9 14 15
51) N* 8 Nlf* 9 14 2 3 7

```
52) N* 8 Nlf* 9 14 2 Nlf* 4 6 3
53) N* 8 Nlf* 9 14 2 Nlf* 4 5
54) N* 8 Nlf* 10 12
55) N* 8 Nlf* 10 11
56) N* 8 7 8


Descritores para o Grafo(11)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

57) N* 11 Nlf* 14 15
58) N* 11 Nlf* 14 2 Nlf* 9 14
59) N* 11 Nlf* 14 2 Nlf* 10 12
60) N* 11 Nlf* 14 2 Nlf* 10 11
61) N* 11 Nlf* 14 2 Nlf* 7 8
62) N* 11 Nlf* 14 2 Nlf* 4 6 3
63) N* 11 Nlf* 14 2 Nlf* 4 5


Descritores para o Grafo(12)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Nt = 3 5 8 11 12 14 15

64) N* 12 Nlf* 14 15
65) N* 12 Nlf* 14 2 Nlf* 9 14
66) N* 12 Nlf* 14 2 Nlf* 10 12
67) N* 12 Nlf* 14 2 Nlf* 10 11
68) N* 12 Nlf* 14 2 Nlf* 7 8
69) N* 12 Nlf* 14 2 Nlf* 4 6 3
70) N* 12 Nlf* 14 2 Nlf* 4 5

Numero Total de Descritores = 70
```

A.2 Informações Dinâmicas

Nesta seção estão organizadas as informações dinâmicas geradas pela POKE-TOOL durante uma sessão de trabalho, relativa ao teste de uma unidade; no caso, da unidade ENTAB.C. Estas informações, a exemplo das informações estáticas,

são armazenadas em arquivos específicos criados pela própria POKE-TOOL. O estado corrente da sessão de trabalho é mantido no arquivo POKELOG.TES. Primeiramente, são apresentadas as entradas e saídas relativas aos testes executados; estas informações são mantidas pela POKE-TOOL em arquivos denominados INPUTi.TES, TECi.TES, OUTPUTi.TES e PATHi.TES, correspondentes às entradas fornecidas como argumentos da linha de chamada, entradas via teclado, resultados obtidos e caminhos executados, respectivamente. Os arquivos relativos à análise de cobertura correspondem ao conjunto de casos de teste inicial (oito casos de teste), ou seja, ao conjunto de casos de teste elaborado inicialmente, de maneira informal e não sistemática, com o objetivo de exercitar os aspectos funcionais da unidade em teste, porém sem a aplicação explícita de uma técnica funcional de elaboração de casos de teste.

Arquivos INPUTi.tes

Não existem no caso da unidade ENTAB.C

Arquivos TECi.TES

Arquivo TEC1.TES

```
col 1    2    34  rest
```

Arquivo TEC2.TES

```
co
```

Arquivo TEC3.TES

```
c
```

Arquivo TEC4.TES

EOF[1]

Arquivo TEC5.TES

NL[2]

Arquivo TEC6.TES

```
c
```

[1] Arquivo vazio.

[2] Arquivo com uma linha em branco.

Arquivo TEC7.TES

```
col 1    2    34  rest
```

Arquivo TEC8.TES

```
x
 x
  x
   x
    x
     x
      x
       x
        x
         x
          x
           x
```

Arquivo OUTPUTi.TES

Arquivo OUTPUT1.TES

```
col 1 2 34 rest
```

Arquivo OUTPUT2.TES

```
co
```

Arquivo OUTPUT3.TES

```
c
```

Arquivo OUTPUT4.TES

Arquivo OUTPUT5.TES

Arquivo OUTPUT6.TES

```
c
```

Arquivo OUTPUT7.TES

```
col 1 2 34 rest
```

Arquivo OUTPUT8.TES

```
 x
  x
   x
x
 x
  x
   x
x
 x
  x
```

Arquivo PATHi.TES

Arquivo PATH1.TES

```
 1
 2  3  7  9  10  12  13  14  2  3
 7  9  10  12  13  14  2  3  7  9
 10  12  13  14  2  3  4  5  6  3
 7  9  10  12  13  14  2  3  4  6
 3  4  6  3  4  5  6  3  7  9
 10  12  13  14  2  3  4  6  3  4
 6  3  4  5  6  3  7  9  10  12
 13  14  2  3  7  9  10  12  13  14
 2  3  4  6  3  4  5  6  3  7
 9  10  12  13  14  2  3  7  9  10
 12  13  14  2  3  7  9  10  12  13
 14  2  3  7  9  10  12  13  14  2
 3  7  9  14  15
```

Arquivo PATH2.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  10  12  13  14  2  3  7  9
14  15
```

Arquivo PATH3.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  14  15
```

Arquivo PATH4.TES

```
1
2  3  7  9  14  15
```

Arquivo PATH5.TES

```
1
2  3  7  9  10  11  13  14  2  3
7  9  14  15
```

Arquivo PATH6.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  10  11  13  14  2  3  7  9
14  15
```

Arquivo PATH7.TES

```
1
2  3  7  9  10  12  13  14  2  3
7  9  10  12  13  14  2  3  7  9
10  12  13  14  2  3  4  5  6  3
7  9  10  12  13  14  2  3  4  6
3  4  6  3  4  5  6  3  7  9
10  12  13  14  2  3  4  6  3  4
6  3  4  5  6  3  7  9  10  12
13  14  2  3  7  9  10  12  13  14
2  3  4  6  3  4  5  6  3  7
```

```
9  10  12  13  14  2  3  7  9  10
12  13  14  2  3  7  9  10  12  13
14  2  3  7  9  10  12  13  14  2
3  7  9  10  11  13  14  2  3  7
9  14  15
```

Arquivo PATH8.TES

```
1
2  3  4  6  3  7  8  7  9  10
12  13  14  2  3  7  9  10  11  13
14  2  3  4  6  3  4  6  3  7
8  7  8  7  9  10  12  13  14  2
3  7  9  10  11  13  14  2  3  4
6  3  4  6  3  4  6  3  7  8
7  8  7  8  7  9  10  12  13  14
2  3  7  9  10  11  13  14  2  3
4  6  3  4  6  3  4  6  3  4
5  6  3  7  9  10  12  13  14  2
3  7  9  10  11  13  14  2  3  4
6  3  4  6  3  4  6  3  4  5
6  3  4  6  3  7  8  7  9  10
12  13  14  2  3  7  9  10  11  13
14  2  3  4  6  3  4  6  3  4
6  3  4  5  6  3  4  6  3  4
6  3  7  8  7  8  7  9  10  12
13  14  2  3  7  9  10  11  13  14
2  3  4  6  3  4  6  3  4  6
3  4  5  6  3  4  6  3  4  6
3  4  6  3  7  8  7  8  7  8
7  9  10  12  13  14  2  3  7  9
10  11  13  14  2  3  4  6  3  4
6  3  4  6  3  4  5  6  3  4
6  3  4  6  3  4  6  3  4  5
6  3  7  9  10  12  13  14  2  3
7  9  10  11  13  14  2  3  4  6
3  4  6  3  4  6  3  4  5  6
3  4  6  3  4  6  3  4  6  3
4  5  6  3  4  6  3  7  8  7
9  10  12  13  14  2  3  7  9  10
11  13  14  2  3  4  6  3  4  6
3  4  6  3  4  5  6  3  4  6
```

```
3  4  6  3  4  6  3  4  5  6
3  4  6  3  4  6  3  7  8  7
8  7  9  10  12  13  14  2  3  7
9  10  11  13  14  2  3  7  9  14
15
```

Arquivo Pokelog.tes

```
Nome do arquivo testado: entab.c
Numero de casos de teste: 8
Numero de avaliacoes do criterio todos-pot-du-caminhos: 8
Numero de avaliacoes do criterio todos-pot-usos: 8
Numero de avaliacoes do criterio todos-pot-usos/du: 8
Tem parametros de entrada: 0
Tem entrada pelo teclado: 1
```

Arquivo PUOUTPUT.TES

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

<1,(14,2),{ col, tabstops }>
<1,(4,5),{ col, tabstops }>
<2,(8,7),{ newcol }>
<2,(7,8),{ newcol }>
<4,(9,14),{ newcol, tabstops }>
<4,(14,15),{ newcol, tabstops }>
<4,(10,11),{ newcol, tabstops }>
<5,(14,15),{ col }>
<5,(2,3),{ col }>
<5,(14,2),{ col }>
<5,(9,14),{ col }>
<5,(10,11),{ col }>
<8,(14,15),{ col }>
<8,(3,7),{ col }>
<8,(6,3),{ col }>
<8,(4,6),{ col }>
<8,(4,5),{ col }>
<8,(14,2),{ col }>
<8,(9,14),{ col }>
<8,(10,11),{ col }>
```

```
<11,(10,12),{ col }>
<11,(10,11),{ col }>
<12,(7,8),{ col }>

Cobertura Total = 71.250000

Media da Cobertura dos Grafo(i) = 70.431721
```

Arquivo EXEC_PU.TES

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS executadas:

<1,(9,14),{ col, tabstops }>
<1,(10,12),{ col, tabstops }>
<1,(14,15),{ col, tabstops }>
<1,(14,15),{ tabstops }>
<1,(14,2),{ tabstops }>
<1,(10,11),{ col, tabstops }>
<1,(8,7),{ tabstops }>
<1,(7,8),{ col, tabstops }>
<1,(4,6),{ col, tabstops }>
<1,(6,3),{ col, tabstops }>
<1,(6,3),{ tabstops }>
<2,(9,14),{ newcol }>
<2,(10,12),{ newcol }>
<2,(14,15),{ newcol }>
<2,(14,2),{ newcol }>
<2,(10,11),{ newcol }>
<2,(3,4),{ newcol }>
<3,(9,14),{ c }>
<3,(10,12),{ c }>
<3,(14,15),{ c }>
<3,(2,3),{ c }>
<3,(14,2),{ c }>
<3,(10,11),{ c }>
<3,(8,7),{ c }>
<3,(7,8),{ c }>
<3,(4,6),{ c }>
<3,(6,3),{ c }>
```

```
<3,(4,5),{ c }>
<4,(4,6),{ newcol, tabstops }>
<4,(10,12),{ newcol, tabstops }>
<4,(2,3),{ tabstops }>
<4,(14,2),{ newcol, tabstops }>
<4,(8,7),{ newcol, tabstops }>
<4,(7,8),{ newcol, tabstops }>
<4,(3,4),{ newcol, tabstops }>
<4,(4,5),{ newcol, tabstops }>
<5,(10,12),{ col }>
<5,(7,8),{ col }>
<5,(4,6),{ col }>
<5,(4,5),{ col }>
<8,(10,12),{ col }>
<8,(7,8),{ col }>
<11,(14,15),{ col }>
<11,(9,14),{ col }>
<11,(7,8),{ col }>
<11,(6,3),{ col }>
<11,(4,6),{ col }>
<11,(4,5),{ col }>
<11,(14,2),{ col }>
<12,(14,15),{ col }>
<12,(9,14),{ col }>
<12,(10,12),{ col }>
<12,(10,11),{ col }>
<12,(6,3),{ col }>
<12,(4,6),{ col }>
<12,(4,5),{ col }>
<12,(14,2),{ col }>

Cobertura Total = 71.250000

Media da Cobertura dos Grafo(i) = 70.431721
```

Arquivo PUDUOUTPUT.TES

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS/DU nao executadas:
```

```
<1,(14,2),{ col, tabstops }>
<1,(8,7),{ tabstops }>
<1,(7,8),{ col, tabstops }>
<1,(4,5),{ col, tabstops }>
<2,(8,7),{ newcol }>
<2,(7,8),{ newcol }>
<4,(9,14),{ newcol, tabstops }>
<4,(14,15),{ newcol, tabstops }>
<4,(10,11),{ newcol, tabstops }>
<5,(14,15),{ col }>
<5,(2,3),{ col }>
<5,(14,2),{ col }>
<5,(9,14),{ col }>
<5,(10,11),{ col }>
<5,(7,8),{ col }>
<5,(4,5),{ col }>
<8,(14,15),{ col }>
<8,(3,7),{ col }>
<8,(6,3),{ col }>
<8,(4,6),{ col }>
<8,(4,5),{ col }>
<8,(14,2),{ col }>
<8,(9,14),{ col }>
<8,(10,11),{ col }>
<11,(14,15),{ col }>
<11,(10,12),{ col }>
<11,(10,11),{ col }>
<11,(7,8),{ col }>
<11,(4,5),{ col }>
<12,(14,15),{ col }>
<12,(7,8),{ col }>

Cobertura Total = 61.250000

Media da Cobertura dos Grafo(i) = 60.175312
```

Arquivo EXEC_PUDU.TES

```
ASSOCIACOES DO CRITERIO TODOS POT-USOS/DU executadas:
```

```
<1,(9,14),{ col, tabstops }>
<1,(10,12),{ col, tabstops }>
<1,(14,15),{ col, tabstops }>
<1,(14,15),{ tabstops }>
<1,(14,2),{ tabstops }>
<1,(10,11),{ col, tabstops }>
<1,(4,6),{ col, tabstops }>
<1,(6,3),{ col, tabstops }>
<1,(6,3),{ tabstops }>
<2,(9,14),{ newcol }>
<2,(10,12),{ newcol }>
<2,(14,15),{ newcol }>
<2,(14,2),{ newcol }>
<2,(10,11),{ newcol }>
<2,(3,4),{ newcol }>
<3,(9,14),{ c }>
<3,(10,12),{ c }>
<3,(14,15),{ c }>
<3,(2,3),{ c }>
<3,(14,2),{ c }>
<3,(10,11),{ c }>
<3,(8,7),{ c }>
<3,(7,8),{ c }>
<3,(4,6),{ c }>
<3,(6,3),{ c }>
<3,(4,5),{ c }>
<4,(4,6),{ newcol, tabstops }>
<4,(10,12),{ newcol, tabstops }>
<4,(2,3),{ tabstops }>
<4,(14,2),{ newcol, tabstops }>
<4,(8,7),{ newcol, tabstops }>
<4,(7,8),{ newcol, tabstops }>
<4,(3,4),{ newcol, tabstops }>
<4,(4,5),{ newcol, tabstops }>
<5,(10,12),{ col }>
<5,(4,6),{ col }>
<8,(10,12),{ col }>
<8,(7,8),{ col }>
<11,(9,14),{ col }>
<11,(6,3),{ col }>
<11,(4,6),{ col }>
```

```
<11,(14,2),{ col }>
<12,(9,14),{ col }>
<12,(10,12),{ col }>
<12,(10,11),{ col }>
<12,(6,3),{ col }>
<12,(4,6),{ col }>
<12,(4,5),{ col }>
<12,(14,2),{ col }>

Cobertura Total = 61.250000

Media da Cobertura dos Grafo(i) = 60.175312
```

Arquivo PDUOUTPUT.TES

```
POTENCIAIS-DU-CAMINHOS que nao foram executados:

Caminhos:
1 2 3 7 9 14 2
1 2 3 7 9 10 12 13 14 15
1 2 3 7 9 10 11 13 14 15
1 2 3 7 8 7
1 2 3 4 5 6 3
2 3 7 9 14 2
2 3 7 9 10 12 13 14 15
2 3 7 9 10 11 13 14 15
2 3 7 8 7
3 7 9 14 2 3
3 7 9 10 12 13 14 15
3 7 9 10 11 13 14 15
4 6 3 7 9 14 15
4 6 3 7 9 14 2 3
4 6 3 7 9 10 12 13 14 15
4 6 3 7 9 10 12 13 14 2 3
4 6 3 7 9 10 11 13 14 15
4 6 3 7 9 10 11 13 14 2 3
4 5 6 3 7 9 14 15
4 5 6 3 7 9 14 2 3
4 5 6 3 7 9 10 12 13 14 15
```

```
4 5 6 3 7 9 10 11 13 14 15
4 5 6 3 7 9 10 11 13 14 2 3
4 5 6 3 7 8 7
5 6 3 7 9 14 15
5 6 3 7 9 14 2 3
5 6 3 7 9 10 11
5 6 3 7 8
5 6 3 4 5
8 7 9 14 15
8 7 9 14 2 3 7
8 7 9 14 2 3 4 6 3
8 7 9 14 2 3 4 5
8 7 9 10 11
11 13 14 15
11 13 14 2 3 7 9 10 12
11 13 14 2 3 7 9 10 11
11 13 14 2 3 7 8
11 13 14 2 3 4 5
12 13 14 15
12 13 14 2 3 7 8

Cobertura Total = 41.428571

Media da Cobertura dos Grafo(i) = 42.906746
```

Arquivo EXEC_PDU.TES

```
POTENCIAIS-DU-CAMINHOS que foram executados:

Caminhos:
1 2 3 7 9 14 15
1 2 3 7 9 10 12 13 14 2
1 2 3 7 9 10 11 13 14 2
1 2 3 4 6 3
2 3 7 9 14 15
2 3 7 9 10 12 13 14 2
2 3 7 9 10 11 13 14 2
2 3 4
3 7 9 14 15
3 7 9 10 12 13 14 2 3
```

```
3 7 9 10 11 13 14 2 3
3 7 8 7
3 4 6 3
3 4 5 6 3
4 6 3 7 8 7
4 6 3 4
4 5 6 3 7 9 10 12 13 14 2 3
4 5 6 3 4
5 6 3 7 9 10 12
5 6 3 4 6
8 7 9 10 12
8 7 8
11 13 14 2 3 7 9 14
11 13 14 2 3 4 6 3
12 13 14 2 3 7 9 14
12 13 14 2 3 7 9 10 12
12 13 14 2 3 7 9 10 11
12 13 14 2 3 4 6 3
12 13 14 2 3 4 5

Cobertura Total = 41.428571

Media da Cobertura dos Grafo(i) = 42.906746
```

Bibliografia

[CAR91] M. Carnassale, *GFC - Uma Ferramenta Multilinguagem para Geração do Grafo de Programa*, DCA/FEE/UNICAMP, Fev., 1991.

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

[CHA91a] M. L. Chaim, J. C. Maldonado e Mario Jino, *Manual de Configuração da POKE-TOOL, Relatório Técnico* DCA/RT/008/91 - DCA/FEE/UNICAMP - Fev. 1991.

[CHA91b] M. L. Chaim, J. C. Maldonado e Mario Jino, *Manual do Usuário da POKE-TOOL, Relatório Técnico* DCA/RT/009/91 - DCA/FEE/UNICAMP - Fev. 1991.

[CHA91c] M. L. Chaim, J. C. Maldonado e Mario Jino, *Projeto de uma Ferramenta de Teste de Programas, Relatório Técnico* DCA/RT/010/91 - DCA/FEE/UNICAMP - Fev. 1991.

[CHU87] T. Chusho, "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing," *IEEE Trans. Software Eng.* Vol. SE - 13, No.5, pp.509-517, Maio 1987.

[DEU82] Deutsch, M. S. *Software verification and validation.* Englewood Cliffs, Prentice-Hall, 1982.

[FRA85] F. G. Frankl e E.J. Weyuker, "Data Flow Testing Tool," in *Proc. Softfair II*, San Francisco, CA, pp.46-53, Dez. 1985.

[FRA87] F. G. Frankl, *The use of data flow information for the selection and evaluation of software test data.* Ph. D. thesis, New York University, 1987.

[FRA88] F. G. Frankl e E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Eng.* Vol. 14, No. 10, pp. 1483-1498, Oct. 1988.

[GHE87] C. Ghezzi e M. Jazayeri, *Programing Languages Concepts.* New York, John Wiley and Sons, Second Edition, 1987.

[HEC77] M. S. Hecht, *Flow Analysis of Computer Programs.* New York, North Holland, 1977.

[HER76] P. M. Herman, "Data Flow Approach to Program Testing," *Australian Computer Journal*, Vol. 8, No.3, Nov. 1976.

[HOR87] S. Horwitz, A. Demers e T. Teitelbaum, "An Efficient General Iterative Algorithm for Dataflow Analysis," *Acta Informatica*, Vol. 24, pp. 679-694, 1987.

[KAM77] J. B. Kam e J. D. Ullman, "Monotone Data Flow Frameworks," *Acta Informatica*, Vol. 7, No. 3, pp. 305-317, 1977.

[KER78] B. W. Kernighan e D. M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[KER81] B. W. Kernighan e P. J. Plauger, *Software Tools in Pascal.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[KOR85] B. Korel e J. W. Laski, "A Tool for Data Flow Oriented Program Testing," *in Proc. IEEE Softfair II*, San Francisco, CA, pp. 34-38, Dez. 1985.

[KOR87] B. Korel, "The Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol. 24, No. 2, pp. 103-108, Jan. 1987.

[LAS83] J. W. Laski e B. Korel, "A Data Flow Oriented Program Testing Strategy," *IEEE Trans. Software. Eng.*, Vol. SE - 9, No.3, pp. 347-354, Maio 1983.

[LUT90] M. Lutz, "Testing Tools," *IEEE Software*, Vol. 7, No. 3, Maio 1990, pp. 57.

[MAL88a] J. C. Maldonado, M. L. Chaim, M. Jino, " Seleção de Casos de Testes Baseada em Fluxo de Dados através dos Critérios Potenciais Usos," *in Proc. II Simp. Bras. Eng. de Software*, Canela, R.S., pp. 24-35, Out. 1988.

[MAL88b] J. C. Maldonado, M. L. Chaim, M. Jino, "Resultados do estudo de uma família de critérios de teste de programas baseado em fluxo de dados," *Relatório Técnico* DCA/RT/001/88 - DCA/FEE/UNICAMP - Dez. 1988.

[MAL89] J. C. Maldonado, M. L. Chaim, M. Jino, "Arquitetura de Uma Ferramenta de Teste de Software de Apoio ao Critérios Potenciais Usos," *in Proc. XXII Congresso Nacional de Informática*, São Paulo, S.P., pp. 92-101, Set. 1989.

[MAL91a] J. C. Maldonado, *Tese de Doutorado*, DCA/FEE/UNICAMP (em andamento), 1991.

[MAL91b] J. C. Maldonado, M. L. Chaim, M. Jino, "Potential Uses Criteria: A Step towards Bridging the Gap in the Presence of Infeasible Paths," submetido para publicação, Março 1991.

[MAL91c] J. C. Maldonado, M. L. Chaim, M. Jino, "Using the Essential Branch Concept to Support Data-flow Based Testing Criteria Application," submetido para publicação, Março 1991.

[MCC76] T. McCabe, "A Software Complexity Measure," *IEEE Trans. Software Eng.*, Vol. 2, pp. 308-320, Dez. 1976.

[MYE79] G. J. Myers, *The Art of Software Testing*. New York, Wiley, 1979.

[NTA84] S. C. Ntafos, "On Required Element Testing," *IEEE Trans. Software Eng.*, Vol. SE - 10, No. 6, pp. 795-803, Nov. 1984.

[NTA88] S. C. Ntafos, "A Comparison of Some Structural Testing Strategies," *IEEE Trans. Software Eng.*, Vol. 14, No. 6, pp. 868-873, Jun. 1988.

[POD90] A. Podgurski e L. A. Clarke, "A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance," *IEEE Trans. Software Eng.*, Vol. 16, No. 9, pp. 965-979, Set. 1990.

[PRE87] R. B. Pressman, *Software Engineering: a Practitioner's Approach*, Second Edition, New York, McGraw-Hill, 1987.

[PRI87] A. M. Price, F. Garcia e C. B. Purper, "Visualizando o Fluxo de Controle de Programas," *in Proc. I Simp. Bras. Eng. de Software*, Petropólis, R.J., pp. 1-9, Out. 1987.

[PRI90] A. M. Price e A. Zorzo, "Ambiente de Apoio ao Teste Estrutural de Programas," *in Proc. IV Simp. Bras. Eng. de Software*, Águas de São Pedro, S.P., pp. 169-182, Out. 1990.

[RAP82] S. Rapps e E. J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection" *in Proc. Int. Conf. Software Eng.*, Tokio, Japão, pp. 272-278, Sept. 1982.

[RAP85] S. Rapps e E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.*, Vol. SE - 11, pp. 367-375, Abril 1985.

[REI90] S. Reisman, "Management and Integrated Tools," *IEEE Software*, Vol. 7, No. 3, pp. 75, Maio 1990.

[SEB89] R. W. Sebesta, *Concepts of Programming Languages.* Benjamin Cummings Publishing Company, Inc.., Redwood City, 1989.

[SET81] V. W. Setzer e I. S. H. de Melo, "A Construção de um Compilador", terceira edição, Editora Campus, Rio de Janeiro, 1981.

[WEY84] E. J. Weyuker, "The Complexity of Data Flow Criteria for Test Data Selection," *Information Processing Letters*, Vol. 19, No.2, Ago. 1984.

[WEY88] E. J. Weyuker, "An Empirical Study of the Complexity of Data Flow Testing," *in Proc. Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, pp. 188-195, Jul. 1988.

[WEY90] E. J. Weyuker, "The Cost of Data Flow Testing: An Empirical Study," *IEEE Trans. Software Eng.*, Vol. 16, No. 2, pp. 121-128, Fev. 1990.

[WIR76] N. Wirth, *Algorithms + Data Structures = Programs.* Englewood Cliffs, NJ, Prentice-Hall, 1976.

[WHI80] L. J. White e E. I. Cohen, "A Domain Strategy for Computer Program Testing," *IEEE Trans. Software Eng.*, Vol. SE - 6, No. 3, pp. 247-257, Maio 1980.

[WHI85] L. J. White e P. N. Sahay, "A Computer System for Generating Test Data Using the Domain Strategy," *in Proc. Softfair II*, San Francisco, CA, pp. 38-45, Dez. 1985.

[URA88] H. Ural e B. Yang, "A Structural Test Selection Criterion," *Information Processing Letters*, Vol. 28, pp. 157-163, Jul. 1988.