POKE-TOOL — Versão 2.0
Manual do Usuário

Paulo M. S. Bueno

Marcos L. Chaim

Mario Jino

José Carlos Maldonado

Plinio R. Vilela

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

1 Prefácio

Este manual foi escrito tendo como objetivo de propiciar um aprendizado mais rápido possível dos recursos básicos da ferramenta. Utilizando esses recursos básicos — após o entendimento de um número mínimo de comandos — o usuário poderá realizar uma sessão completa de teste em seus programas, segundo critérios solidamente embasados no seu aspecto conceitual.

Após ter assimilado as características gerais do funcionamento da ferramenta, o usuário pode analisar de maneira mais atenta o potencial de cada comando, concentrando a atenção na sintaxe dos mesmos, nos parâmetros aceitos e nas informações geradas no vídeo e em arquivos.

Adota-se as seguintes convenções tipográficas:

Nomes de arquivos e de comandos presentes no texto, assim como os títulos e subtítulos encontram-se em

negrito;

Comandos e mensagens emitidos no vídeo estão em fonte típica de

```
maquina de escrever;
```

Em outras situações, quando deseja-se destacar nomes, utiliza-se fonte em

itálico.

A Seção 1 é uma introdução em que se apresenta a ferramenta, suas principais características e destaca-se a evolução conseguida em relação às versões anteriores. São feitos breves comentários sobre a importância do teste de software dentro do contexto da engenharia de software, e de como a **poketool** apóia tal processo.

A seção de teste básica mostrada na Seção 2 permite que se aprenda os aspectos essenciais do processo de teste apoiado pela ferramenta.

A Seção 3 tem como finalidade fornecer auxílio na compreensão do funcionamento de cada comando, e também na interpretação e utilização das informações geradas em cada etapa do teste.

Indicam-se referências bibliográficas para o usuário interessado em se aprofundar no aspecto teórico do processo de teste e dos critérios apoiados pela ferramenta.

O procedimento para a instalação encontra-se no apêndice A.

O usuário já familializado com o processo de teste usando a **poketool**, pode dirigir-se diretamente à Seção 3, utilizando-a como referência para esclarecer dúvidas relativas à sintaxe à e semântica dos comandos.

Para assimilar melhor os conceitos fundamentais, aconselha-se seguir cuidadosamente os passos presentes na Seção 2, executando os comandos e analisando os resultados produzidos.

As Seções 2 e 3 não são excludentes, mas sim complementares, visto que elas enfocam a utilização da **poketool** de pontos de vistas distintos.

Conteúdo

Lista de Figuras

2 Introdução

Uma atividade essencial para o aumento da qualidade de software é a atividade de teste. No teste de software executa-se o programa com intuito de revelar os defeitos nele existentes que provocam desvios no comportamento observado do programa em relação ao comportamento esperado, presente na especificação do software.

Quando o processo de teste é eficazmente realizado, tem-se uma boa chance de revelar grande parte dos defeitos e de aumentar a confiabilidade do programa.

Para que se tenha um processo de teste eficaz, é essencial utilizar critérios e metodologias que o dirija. O problema essencial é conseguir a adequada seleção de dados de teste, assim como uma avaliação objetiva da qualidade do teste efetuado, através de algum parâmetro confiável.

A utilização de critérios de teste de forma não automatizada em ambientes profissionais é praticamente impossível. Tal fato estimula a utilização de ferramentas que permitam uma atividade de teste de boa qualidade, a um custo compatível com a realidade, em termos de tempo e esforço dispendidos.

A **poketool** (POtencial uses CRIteria TOOL for program testing) constitui-se em uma ferramenta interativa que visa auxiliar o usuário no processo de teste de software [2].

Tal ferramenta propicia a aplicação de critérios de teste de software que se baseiam em informações do código do programa, para a seleção de dados de teste e para a análise da cobertura atingida (relacionada ao nível de completitude do teste).

Por utilizarem informações relativas à estrutura interna do programa, tais critérios são chamados *estruturais*, e têm um enfoque comum. Todos critérios implementados na ferramenta buscam a caracterização de um conjunto de componentes elementares de um programa (chamados elementos requeridos) que devem ser exercitados através dos casos de teste aplicados, durante o processo de teste do programa.

A **poketool** apóia a aplicação dos critérios estruturais baseados em fluxo de controle *todos nós* e *todos arcos*, assim como os *critérios Potenciais Usos* (PU) [8], os quais são baseados em análise de fluxo de dados [5].

Os critérios *todos nós* e *todos arcos* requerem respectivamente: que todos nós executáveis do grafo do programa sejam executados; e que todos os arcos executáveis do programa em teste sejam executados.

Os critérios baseados em fluxo de dados [4], [7], [10] requerem que as associações que envolvem definições de variáveis de programa e subseqüentes referências a estas definições sejam testadas. Mais especificamente, a família de *critérios Potenciais Usos* [9], que é constituída dos critérios: *Todos Potenciais DU Caminhos*; *Todos Potenciais Usos* e *Todos Potenciais Usos/DU*, requer associações independentemente da ocorrência explícita de uma referência a uma determinada definição. Ou seja, se o uso de uma definição pode existir, a respectiva *potencial associação* é requerida. No contexto deste manual, a *potencial associação* será referida apenas como *associação*.

A implementação inicial de **poketool** foi desenvolvida em micros compatíveis com IBM/PC PC. O objetivo a princípio era o de implementar uma ferramenta específica para o auxílio ao teste aplicando os critérios PU, permitindo, portanto, a avaliação da eficácia desta família de critérios e da sua relação com outros já existentes. A interação com o usuário se dava através de uma seqüência de telas que eram formadas por menus e por um campo de mensagens. Através da seleção de opções no menu o usuário acionava programas e visualizava arquivos gerados. Esta primeira versão, entretanto, possibilitava o teste de apenas uma unidade (função na linguagem C) por vez, o que tornava dispendioso o teste de diversas unidades. Além disto, o sistema de interface utilizando menus, apesar de tornar a ferramenta mais cognitiva, por embutir conceitos de utilização na própria interface, apresentava-se pouco produtivo, considerando-se os requisitos da atividade de teste em ambientes profissionais.

Tendo em vista as necessidades deste tipo de ambiente de teste, no qual a eficiência é um fator preponderante, surgiu a versão 2.0 da **poketool**.

A versão atual está disponível para utlilização em estações SUN/UNIX. Neste ambiente, o processo de teste é apoiado por programas distintos que permitem a análise estática do programa em teste, a execução de casos de teste, e a avaliação da cobertura atingida. É permitida também a visualização do grafo do programa. Nas próximas versões, novas funcionalidades, como o auxílio ao tratamento de associações e caminhos não executáveis, serão integradas à **poketool**.

A característica princiapal da ferramenta, sob o aspecto da interação com o usuário, diz respeito ao funcionamento por linhas de comando. Cada fase do proceso de teste é apoiada por um programa distinto, acionado diretamente pelo testador através do comando correspondente. Cada programa recebe como entrada uma série de arquivos gerados em etapas anteriores, além de informações fornecidas pelo usuário. São produzidas mensagens e ralatórios no vídeo e em arquivos.

A interação com a ferramenta, feita por linhas de comando, possibilita uma maior eficiência, devido a maior objetividade com que se realiza as tarefas inerentes ao processo de teste. Além disto, a versão atual é compatível com declarações de funções no estilo ANSI e constitui-se ainda em uma plataforma inicial para a adoção de uma interface padrão *Motif*, a ser desenvolvida como uma versão alternativa à atual.

Um aspecto relevante consiste na possibilidade de se testar diversas unidades simultaneamente. Considerando como *unidade* uma função em C que está sendo submetida à **poketool**, na atual versão, pode-se realizar a análise estática de todas unidades de um arquivo, executar casos de teste que as exercitem, e avaliar a cobertura atingida para cada unidade distinta, de acordo com o critério escolhido. Esta é uma característica que a versão anterior da ferramenta não possuía.

Na seção seguinte, introduzem-se conceitos básicos da ferramenta; na Seção 3, são detalhados aspectos da sintaxe e semântica dos comandos. A descrição do procedimento de instalação da ferramenta está no **apêndice A**. Incluem-se no final do manual, apêndices que mostram listagens e detalham os arquivos gerados pela **poketool**.

3 Utilização Básica

Na seção que se segue, serão mostrados e ilustrados os passos básicos para o teste de um programa utilizando a **poketool**. O objetivo desta seção é permitir que o testador tenha uma visão geral da ferramenta e que aprenda a utilizar os comandos existentes em sua forma básica. Sugere-se, portanto, que a atenção seja direcionada para a seqüência das tarefas descritas, bem como para o fluxo das informações geradas e utilizadas pela ferramenta. Uma visão pormenorizada dos comandos, com a explicação detalhada da sintaxe, parâmetros aceitos e arquivos gerados está presente na Seção 4.

Como já dito, a **poketool** apóia o teste estrutural de funções escritas na linguagem C segundo diversos critérios baseados em informações do fluxo de dados e de controle. Pode-se utilizar a ferramenta para testar funções de maneira isolada, ou testá-las integradas entre si e a outros módulos. A seguir é descrito o programa exemplo — denominado **find.c** — a ser utilizado no manual.

3.1 Programa Exemplo

O programa recebe como entrada, nesta ordem:

- O número de elementos de um vetor de números inteiros;
- A posição do elemento a ser utilizado como "pivô";
- O valor de cada elemento do vetor.

A partir desses valores, é gerado um vetor de saída, com o mesmo número de elementos do vetor de entrada no qual todos elementos menores ou iguais ao valor do elemento pivô são posicionados à esquerda deste. Os elementos cujos valores forem maiores que o elemento pivô são posicionados à direita do mesmo.

A listagem do programa encontra-se no **apêndice B**.

O programa fonte encontra-se no disquete anexo. Para um melhor aprendizado do funcionamento básico da ferramenta, aconselha-se que o usuário execute os comandos relativos a cada fase do teste, acompanhando as mensagens emitidas e as informações geradas.

Vale destacar que ao digitar qualquer comando, sem fornecer nenhum parâmetro, serão mostradas informações sobre como utilizá-lo. Tal recurso deve ser utilizado sempre que necessário.

3.2 Criando diretório de trabalho

Após feita a instalação da ferramenta (Apêndice A), deve-se criar um diretório que será utilizado no processo de teste; nesta seção utilizaremos o diretório **/test**. Inicialmente deve-se copiar o programa a ser testado, **find.c**, para o diretório **/test**. O acionamento dos comandos, exceto quando dito o contrário, deve ser feito a partir deste diretório de trabalho.

3.3 PASSO 1. Realizando a análise estática

A primeira etapa do teste utilizando a **poketool** tem como finalidade extrair informações descritivas do programa e prepará-lo para a fase dinâmica, a ser realizada posteriormente, e que consiste na execução dos casos de teste. Deve-se executar o seguinte comando:

Comando 1:

```
poketool find.c -ffind
```

Serão emitidas as seguintes mensagens que informam sobre a execução das funções relacionadas ao programa **poketool**.

```
poketool - Analisador Estatico para Teste de Programas escritos em Linguagem C
Arquivo em teste: find.c
* * Carregando Tabela de transicao lexica... * *
* * Carregando Tabela de palavras chaves... * *
* * Carregando Parser especifico da linguagem fonte... * *
* * Fazendo a analise sintatica do codigo fonte... * *
* * Nucleo POKETOOL foi bem sucedido * *

Gerando os descritores para a funcao find ...
* * Calculando os arcos primitivos... * *
* * Gerando descritores... * *
* * Geracao de Descritores da POKETOOL foi bem sucedida * *
* * Gerados os descritores da funcao find * *
```

O **comando 1** aciona o programa **poketool**. O primeiro parâmetro refere-se ao nome do arquivo **.c** no qual está a função a ser testada; o segundo parâmetro informa o nome da função específica que será testada. Deve-se observar que no arquivo **find.c** além da função **find**, existe a função **main**. Caso se desejasse testar também essa função, o comando correto seria:

`poketool find.c -ffind -fmain.`

Neste caso seriam geradas informações estáticas também da função **main** que seria testada de forma integrada à **find**.

Durante a execução do **comando 1**, o programa **poketool** lê o programa em teste, faz a análise estática do mesmo e gera informações descritivas — além do programa modificado para teste.

Com relação à função **find**, as informações descritivas estão em arquivos localizados no subdiretório **/find**, gerado pela ferramenta, e dizem respeito aos elementos do programa que deverão ser exercitados durante a fase dinâmica a ser realizada posteriormente.

Para visualizar tais informações, o testador deve deslocar-se para o diretório respectivo e utilizar os utilitários *cat* ou *more* do UNIX. Pode-se acionar por exemplo: `more pdupath.tes` para visualizar a descrição dos *potenciais-du-caminhos* [9] existentes na função **find**, ou `more puassoc.tes` para visualizar as associações presentes nesta função.

O programa modificado para teste é gerado no diretório de trabalho (**/test**) com o nome **testeprog.c** e consiste em um programa funcionalmente equivalente ao programa original, exceto pela adição de *pontas de prova* (comandos de escrita em arquivo), que permitem a obtenção do caminho executado por caso de teste no grafo do programa. Deve-se notar que apenas a função a ser testada (**find**, no caso) é instrumentada, permanecendo as demais do arquivo, caso existam, inalteradas.

Após executado o programa **poketool** passa-se para a fase dinâmica do teste. Todavia, para a execução de casos de teste, antes é necessário compilar o programa **testeprog.c**

3.4 PASSO 2. Compilando o programa instrumentado

Deve-se agora utilizar um compilador C disponível no ambiente UNIX/SUN para compilar o programa **testeprog.c** gerado pelo programa **poketool**. Pode-se acionar por exemplo o comando:

Comando 2:

```
gcc testeprog.c -o testeprog
```

Tal comando gera o programa executável **testeprog** que será utilizado no passo seguinte.

3.5 PASSO 3. Executando casos de teste

Nesta etapa são executados os casos de teste. A elaboração dos casos de teste pode ser baseada em técnicas funcionais, estruturais ou mesmo de maneira *ad hoc*. Para executá-los, deve-se utilizar o programa **pokeexec** que auxilia no armazenamento de informações importantes como o caminho percorrido, as entradas dos casos de teste, bem como a saída obtida. O **pokeexec** recebe as seguintes informações:

1. nome do programa executável;
2. tipo de entrada dos dados (parâmetros e/ou entradas pelo teclado);
3. se a sessão é contiuação de uma sessão anterior; e
4. nome da função em teste;

Para testar o programa compilado no passo anterior deve-se executar o seguinte comando:

Comando 3:

```
pokeexec testeprog -t -ffind
```

O primeiro parâmetro consiste no nome do programa executável que será testado; **-t** informa que o programa aceita entradas pelo teclado. Caso ele aceitasse também parâmetros na linha de comando, seria necessário acrescentar **-l** após o **-t** . Por último, **-ffind** especifica o nome da função que será testada.

Após a execução do comando 3, são emitidas as seguintes mensagens:

```
pokeexec mensagem: Comecando sessao de teste para o programa
 "testeprog" ...

pokeexec mensagem: removendo todos os arquivos do diretorio
 "output"
pokeexec: "testeprog" aceita parametros em linha de comando
 ? (s/n) [s] n
pokeexec mensagem: removendo todos os arquivos do diretorio
 "keyboard"
pokeexec:  Executando caso de teste numero 1.
```

Neste ponto podem ser introduzidos os casos de teste. O **pokeexec** aciona o programa em teste, o que possibilita a entrada de dados, o processamento das informações e a saída dos resultados. Note-se que no final de cada caso de teste, deve-se informar se se deseja executar mais casos de teste ou não.

Podem ser executados os seguintes casos de teste:

Observação 1: O primeiro dado refere-se ao número de elementos do vetor, o segundo à posição do pivô, os demais são os elementos do vetor.

Observação 2: Após cada dado deve-se digitar <ENTER>. Terminada a entrada de dados, deve-se digitar o caractere CTRL-D para encerrar a entrada de dados via teclado. Este caractere é sempre utilizado como o finalizador da entrada via teclado.

Caso 1:

Entrada : 5 3 15 10 20 45 80
Saída Esperada : 5 3 15 10 20 45 80

Caso 2:

Entrada : 0 0
Saída Esperada : 0 0

Caso 3:

Entrada : 3 3 10 20 30
Saída Esperada : 3 3 10 20 30

Caso 4:

Entrada : 3 3 30 20 10
Saída Esperada : 3 3 10 20 30

Neste ponto do processo de teste, deve-se checar se a saída obtida corresponde à esperada, definida na especificação do software. Naturalmente, caso o programa falhe, o defeito causador deve ser identificado e corrigido. Nesta situação, o teste deve ser reiniciado do **passo 1** .

Terminada a execução do último caso de teste, pode-se visualizar algumas informações geradas nesta etapa.

É criado pela ferramenta um subdiretório **/test/keyboard** no qual estão armazenadas as entradas pelo teclado dos dados de teste executados. Analogamente, é criado um diretório **/test/output**, onde estão as saídas produzidas para os mesmos. No diretório **/find** estão armazenados os caminhos produzidos por cada caso de teste executado (arquivos **path.tes**).

O passo seguinte consiste na avaliação do teste realizado.

3.6 PASSO 4. Avaliando a cobertura atingida

O objetivo desta etapa é avaliar a qualidade do conjunto de teste a que foi submetido o programa. Para tanto, deve-se avaliar a cobertura conseguida segundo algum critério de adequação. Esta tarefa é apoiada pelo programa **pokeaval**.

O **pokeaval** recebe os elementos requeridos gerados pelo programa **poketool** e os caminhos executados pelos casos de teste, gerados pelo **pokeexec**. De posse dessas informações são computados quais os elementos requeridos pelo critério já foram exercitados e quais não.

Para fazer a avaliação do teste realizado na função **find** necessita-se acionar o seguinte comando:

Comando 4:

```
pokeaval -dfind -pu -ne 1 to 4
```

O primeiro parâmetro informa o nome do diretório (criado pelo programa **poketool**) da função a ser avaliada; **-pu** seleciona o critério *todos potenciais usos* como o critério de adequação a ser utilizado na avaliação; **-ne** faz com que sejam mostradas as associações não exercitadas pelos casos de teste; **1 to 4** determina que a avaliação deve ser feita considerando-se o caso de teste número 1 até o caso de teste número 4 (incluindo os extremos).

É importante notar que poderiam ser utilizados outros critérios na avaliação do teste, no entanto, para cada critério, deve-se executar novamente o **pokeaval**.

Após executado o **comando 4** serão emitidas as seguintes mensagens que dizem respeito ao funcionamento do **pokeaval**:

```
pokeaval - Avaliador de Casos de Teste da POKE-TOOL
pokeaval mensagem: removendo todos os arquivos do diretorio "goodpu"

**** Avaliando Caso de Teste Numero 1 ****

* * Realizando a avaliacao do caso de teste * *

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

**** Avaliando Caso de Teste Numero 2 ****
```

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

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

**** Avaliando Caso de Teste Numero 3 ****

* * Realizando a avaliacao do caso de teste * *

* * Avaliacao do caso de teste foi bem sucedida * *
pokeaval: Msg: Caso de teste 3 NAO matou nenhuma associacao!

**** Avaliando Caso de Teste Numero 4 ****

* * Realizando a avaliacao do caso de teste * *

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

ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

<1,(3,5),{ b, m, ns, a }>
<1,(18,20),{ m, ns, a }>
<1,(18,20),{ b, m, ns, a }>
```

Obs: Listagem resumida para apresentação.

```
<20,(13,21),{ b }>
<20,(8,9),{ b }>
<20,(3,4),{ b }>

Cobertura Total = 30.625000

Media da Cobertura dos Grafo(i) = 27.415195
```

Além da informação mostrada no vídeo, que relata os elementos requeridos ainda não exercitados e a cobertura atingida, são gerados arquivos no subdiretório **/find**, com informações relativas à avaliação realizada. Após a execução do **comando 4** serão criados os arquivos:

exec_pu.tes — mostra as associações já exercitadas;

puoutput.tes — lista as ainda não exercitadas;

puhis.tes — armazena o histórico das avaliações realizadas até o momento, de tal sorte que, quando uma nova avaliação é iniciada, o avaliador da **poketool** consulta este arquivo para saber quais elementos requeridos do critério já foram satisfeitos por avaliações prévias.

É criado um subdiretório **/find/goodpu** onde estão arquivos que armazenam as associações cobertas adicionalmente por cada caso de teste, isto é, se o caso de teste 3 cobriu alguma associação adicional em relação aos casos de teste anteriores, então é criado um arquivo **pudif3-2** que contém o número das associações adicionalmente mortas.

Visualizando estes arquivos e analisando seus conteúdos, é possível obter uma série de informações importantes, por exemplo:

Listagem das associações ainda não foram exercitadas. Partindo dessa listagem, do programa instrumentado (**testeprog.c**) e do grafo de fluxo de controle do programa (explica-se em breve como obtê-lo), pode-se criar casos de teste, visando exercitar estas associações que faltam.

Quais associações já foram exercitadas e qual o nível de cobertura conseguido no teste. Por exemplo, foram exercitadas 30,625 % do total de associações existentes na função **find**.

Quais associações cobertas adicionalmente por cada caso de teste. Exemplo: o caso de teste número 2 exercitou adicionalmente apenas a associação número 1 da função **find** (número de referência no arquivo **puassoc.tes**).

Quais casos de teste propiciaram um incremento no nível de cobertura. Exemplo: não foi gerado nenhum arquivo **pdudif3-2** no subdiretório **/find/goodpu**. Isto permite concluir que o caso de teste número 3 não exercitou nenhuma associação adicional e, portanto, não contribuiu para a cobertura atingida do critério de avalição utilizado. Tal fato é reportado pela ferramenta através da mensagem: "pokeaval: Msg: Caso de teste 3 NAO matou nenhuma associacao!".

Caso se deseje analisar a cobertura conseguida com cada caso de teste específico, pode-se acionar o **pokeaval** passando o número de tal caso de teste como limite inferior e superior para avaliação.

Por exemplo:

Comando 5:

```
pokeaval -dfind -pu -e 2 to 2
```

O **comando 5** avalia a cobertura conseguida na execução do caso de teste número 2, em relação ao critério *todos potenciais usos*, considerando as informações sobre a função **find**, contidas no diretório homônimo. É fornecida na tela a listagem das associações executadas e a cobertura atingida (20,00 % do total de associações).

Pode-se repetir o processo de avaliação para cada um dos critérios apoiados pela ferramenta. Por exemplo, para avaliar o teste segundo o critério *todos potenciais du-caminhos*, deve-se acionar o comando:

pokeaval -dfind -pdu -ne 1 to 4.

3.7 PASSO 5. Visualizando o grafo do programa

A visualização do grafo do programa ajuda intensamente no entendimento do mesmo. Tal entendimento é essencial no passo anterior e também no passo a ser apresentado a seguir. Para visualizar o grafo da função **find**, deve-se acionar o seguinte comando, a partir do subdiretório homônimo:

Comando 6:

```
viewgraph find.gfc &
```

Para encerrar o programa **viewgraph**, basta fechar a janela que apresenta o grafo de fluxo de controle.

O arquivo **find.gfc**, passado como parâmetro, consiste em uma descrição textual do grafo de fluxo de controle da função **find** que foi gerada pelo programa **poketool**.

Para terminar sua execução, deve-se fechar a janela que apresenta o grafo.

3.8 PASSO 6. Melhorando o conjunto de casos de teste

Após avaliar a cobertura atingida, o testador pode ter o interesse em aumentar o nível de cobertura, através da adição de mais casos de teste. O programa **pokeexec** permite esta atividade.

A seleção dos casos de teste que irão permitir um acréscimo na cobertura é o ponto crítico desta etapa. A idéia básica consiste em verificar quais dos elementos requeridos pela aplicação do critério ao programa ainda não foram cobertos (informação fornecida pelo **pokeaval**), e então selecionar dados de teste visando exercitá-los.

Para adicionar casos de teste para a função **find**, deve-se acionar o programa **pokeexec**, passando os seguintes parâmetros:

Comando 7:

```
pokeexec testeprog -t -r5 -ffind
```

Este comando aciona o **pokeexec** de forma análoga à descrita no **passo 3**, exceto pelo parâmetro **-r5**, que indica que o teste deve começar a partir do caso de teste número 5. Isto permite que se mantenha as informações geradas anteriormente com os casos de teste 1 a 4. Deste modo, consegue-se que a atual execução da função **find** represente um complemento ao teste já efetuado **no passo 3**. Pode-se executar o seguinte caso de teste:

Caso 5:

Entrada : 7 5 5 5 5 5 8 4 4
Saída Esperada : 7 5 4 5 5 5 5 4 8

Após executá-lo, pode-se observar que no subdiretório **/find** foi criado o arquivo **path5.tes** que contém o caminho efetuado no grafo do programa por este caso de teste. Observe-se que os arquivos **path1.tes** a **path4.tes** foram mantidos, o que possibilita a avaliação do processo de teste como um todo.

É possível agora avaliar o conjunto de teste melhorado utilizando-se o programa **pokeaval**:

Comando 8:

```
pokeaval -dfind -pu -ne 1 to 5
```

Este comando é análogo ao acionado no **passo 4**, exeto pelo parâmetro **1 to 5**, que estende o escopo da avaliação ao caso de teste 5. As informações produzidas também já foram descritas. Cabe destacar o incremento produzido no nível de cobertura para o critério *todos potenciais usos* que de 30,625 % passou a 45,00 %. No arquivo **pudif5-4.tes** estão os números (em relação ao arquivo **puassoc.tes**) dos elementos requeridos cobertos adicionalmente pelo caso de teste número 5.

Pode-se utilizar o **pokeexec** quantas vezes forem necessárias para introduzir novos casos de teste no conjunto inicial, sempre tomando o cuidado de passar como parâmetro **-rnum**, sendo que **num** é o número inteiro imediatamente superior ao número do último caso de teste introduzido anteriormente.

3.9 PASSO 7. Identificando elementos requeridos não executáveis

Depois de devidamente complementado, o conjunto de dados de teste deve ser capaz de alcançar um nível de cobertura relativamente alto; no entanto, a cobertura de 100 %, para a maioria dos critérios apoiados pela ferramenta, não é atingida no **passo 5**. Isto se deve em parte devido a existência de elementos requeridos não executáveis; fato que ocorre na grande maioria dos programas existentes. Para estes elementos, não existe nenhum conjunto de valores de entrada capaz de executá-los.

O **passo 6** consiste, portanto, na identificação de quais elementos requeridos ainda não exercitados são executáveis, e quais não o são. Esta tarefa deve ser realizada manualmente, através da inspeção do código do programa instrumentado (**testeprog.c**). A visualização do grafo do programa (**passo 5**) pode ser útil nesta etapa.

Tomando-se como exemplo a função **find**: a associação $<$*1,(3,5), b, m, ns, a* $>$, segundo elemento requerido presente no arquivo **puassoc.tes**, é uma associação não executável. Isto porque sempre que o programa atingir o comando `if(!b)` no nó 3 com os valores das variáveis *b*,*m*,*ns*,*a* atribuídos no nó 1, o arco (3,4) — e não (3,5) — será executado, pois, neste ponto, *b* possui sempre o valor 0 (atribuído no nó 1). Como conseqüência, a associação referida é não executável.

Atinge-se 100 % de cobertura para um dado critério se todos os elementos requeridos pelo critério, para o programa em teste, forem exercitados; excetuando-se naturalmente os identificados como sendo não executáveis.

A questão da não executabilidade é detalhadamente discutida em [11].

3.10 Diagrama do fluxo de informanções

A Figura 1 ilustra o fluxo de informações entre os vários programas que constituem a **poketool**.

Na seção seguinte tem-se uma descrição aprofundada dos comandos, enfatizando a sintaxe dos mesmos, opções de utilização, e descrevendo de forma pormenorizada os arquivos gerados.

4 Como Utilizar a poketool

Nesta seção são mostradas, de forma mais detalhada, as características da ferramenta. Enfoca-se principalmente a sintaxe de cada comando, assim como os parâmetros aceitos e as informações geradas pelo mesmo. Encontra-se a seguir uma descrição da linguagem aceita por esta versão e também um rápido comentário sobre o modelo de fluxo de dados adotado pela ferramenta.

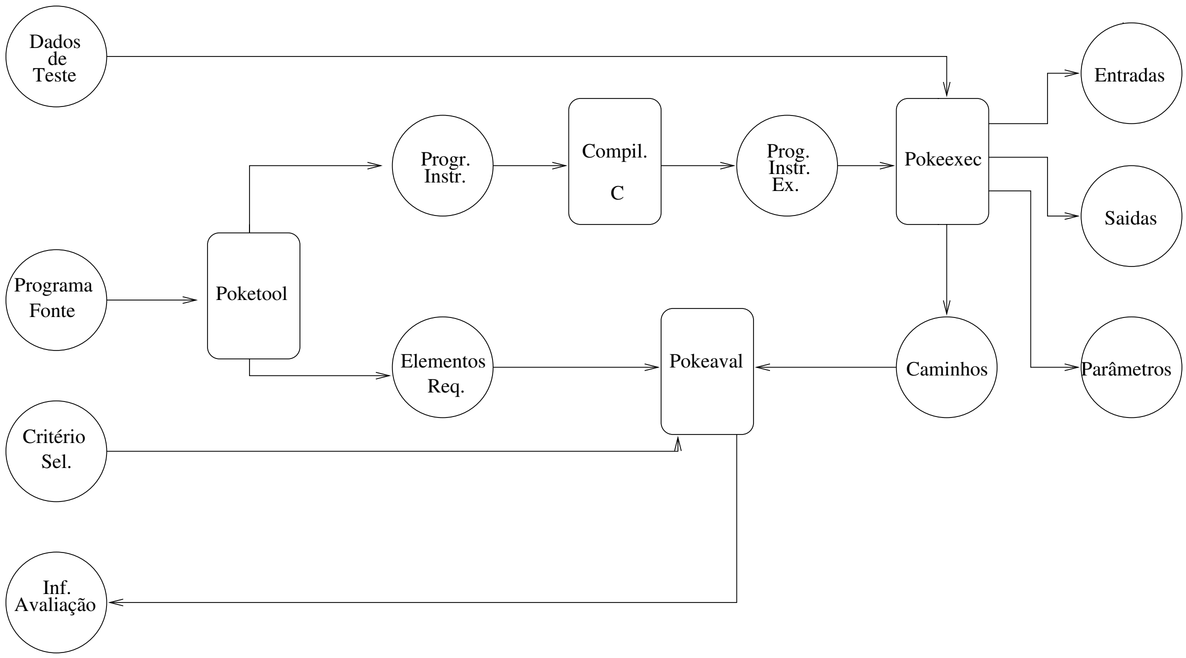

Figura 1: **Fluxo de Informações**

4.1 Linguagem Aceita

A **poketool** aceita programas sintaticamente corretos escritos em C segundo o padrão de Kernighan e Ritchie[1] [6] com as seguintes limitações:

(1) o único comando do pré-processador que pode ser incluído no arquivo que contém a unidade em teste é **#include <nome do arquivo>**;

(2) o número máximo de variáveis na unidade em teste é 159;

(3) o número máximo de blocos de comandos seqüenciais (nós) na unidade em teste é 150;

(4) os identificadores **path**, **printed_nodes** e **ponta_de_prova** não devem ser utilizados, pois são usados no programa modificado obtido a partir da unidade em teste.

O modelo de fluxo de dados adotado na ferramenta inclui os seguintes conceitos:

A definição de variáveis ocorre nas seguintes situações:

(1) A variável está no lado esquerdo de um comando de atribuição;

(2) A variável está em um comando de entrada de dados;

(3) A variável está em chamadas de procedimento como parâmetro de saída.

Variáveis estruturadas (vetores e matrizes) e registros são tratados de forma que a definição de um elemento da variável implica a definição da variável como um todo;

[1] A **poketool** aceita declarações de funções segundo o padrão ANSI, apesar de não suportá-lo completamente.

Variáveis do tipo ponteiro são tratadas exatamente como variáveis comuns, não sendo feita a análise de definições por derreferenciação;

Os seguintes nomes de arquivos não devem ser usados nos diretórios onde vai ocorrer a sessão de trabalho da **poketool**, pois a ferramenta vai gerar arquivos com esses nomes:

> **testeprog.c, testeprog, arcprim.tes, grafodef.tes, des_pdu.tes, des_arc.tes, des_nos.tes, des_pudu.tes, nos_grf.tes, puassoc.tes, tabvardef.tes pdupaths.tes, puout.tes, pduout.tes, puhis.tes, pduhis.tes, output.tes e path.tes.**

Deve-se submeter à **poketool** unidades sintaticamente corretas. Ou seja, é necessário compilar a unidade a ser testada antes de submetê-la à ferramenta.

4.2 Aspectos Gerais

A **poketool** é constituída de quatro programas distintos que permitem a análise estática do programa em teste, a execução de casos de teste e a avaliação da cobertura conseguida.

A **poketool** é uma ferramenta orientada por linhas de comando de tal forma que o usuário, através da execução de um conjunto de comandos, gera um novo programa executável alterado para teste, submete casos de teste, avalia-os e visualiza o grafo da unidade. Tais comandos, quando corretamente executados, acionam os respectivos programas que recebem parâmetros específicos e geram mensagens pertinentes. As informações são armazenadas em arquivos gerados pela **poketool**, tanto para a utilização posterior pela própria ferramenta, quanto para acesso por parte do testador.

As mensagens enviadas pela **poketool** podem ser de dois tipos. Um primeiro tipo são aquelas que indicam o andamento de uma atividade (*e.g.*: "gerando descritores ...") ou a conclusão de uma atividade (*e.g.*: "avaliação dos casos de teste bem sucedida"); outro tipo são as mensagens de erro que indicam a ocorrência de um problema em alguma atividade realizada pela **poketool**.

Um aspecto interessante da atual versão consiste na capacidade de auxiliar o teste de diversas unidades simultaneamente. Neste caso, informações relativas às diferentes unidades são armazenadas em arquivos que pertencem a subdiretórios criados pela ferramenta com o nome da respectiva unidade.

Pode-se definir, em linhas gerais, quatro etapas distintas a serem cumpridas no processo de teste dentro do contexto da ferramenta:

(1) Execução do programa **poketool** — gera informações estáticas e o programa modificado para teste (**testeprog.c**);

(2) Compilação do programa modificado para teste;

(3) Execução do programa modificado, através do programa **pokeexec**, utilizando um conjunto de casos de teste; e

(4) Avaliação do teste realizado, através do programa **pokeaval**, segundo o critério de adequação desejado.

Em seguida, serão detalhados aspectos de cada programa separadamente. Para cadaum deles, destacam-se:

Objetivo — para que serve;

Sinopse — como acioná-lo;

Parâmetros — significado dos parâmetros aceitos;

Funcionamento — mensagens emitidas e informações geradas;

Exemplo de Utilização — exemplo ilustrativo para maior entendimento.

Os arquivos gerados pelos programas encontram-se listados no **apêndice C**.

4.3 Programa Poketool

Objetivo: Realizar a análise estática do programa a ser testado, gerar arquivo modificado para teste e o grafo textual da(s) função (funções) em teste.

Sinopse :

poketool <nome do arquivo> [-I <diretórios de include>] [-D <defines>] -f <função> [-f <função>]

Parâmetros:

- <**nome do arquivo**>: Nome do arquivo com extensão *.c* no qual está(ão) a (as) unidade(s) a ser(em) testada(s).
- **-I** <**diretórios de include**>: Determina os diretórios incluídos no programa em teste;
- **-D** <**defines**>: Opção não implementada;
- **-f** <**função**>: Nome da função a ser testada. Caso se deseje testar mais de uma função silmultaneamente, deve-se especificá-las. Para testar todas as funções sem nomeá-las, deve-se utilizar o nome **-f_todas_** como parâmetro.

Funcionamento:

Ao se executar o programa **poketool**, fornecendo o conjunto de parâmetros corretos, tem-se a geração dos elementos requeridos relativos aos critérios considerados pela ferramenta para todas as funções especificadas como parâmetro. Além disto, são geradas informações relativas à estrutura do programa e uma versão modificada do mesmo.

No diretório corrente são gerados os seguintes arquivos:

- **testeprog.c**: armazena o programa em teste devidamente modificado. Todas as unidades que serão testadas contêm *pontas de prova*, que consistem em comandos de escrita em arquivo cujo objetivo é permitir o registro do caminho percorrido no programa para cada caso de teste;

<função>.li e **<função.nli>**: tais arquivos contêm o programa em teste representado em uma *linguagem intermediária* [1], criada pela **poketool**. São arquivos intermediários utilizados pela ferramenta.

Caso alguma função do arquivo em teste não seja declarada na linha de comando, essa função não será instrumentada para teste (e, obviamente, não será testada). Neste caso, para esta função será gerado (no subdiretório corrente) um arquivo **<função>.gfc** que contém o grafo textual respectivo.

O programa **poketool** gera um subdiretório para cada função instrumentada (passada como parâmetro) do programa em teste. Neste subdiretório são criados arquivos que possuem informações estáticas sobre esta função específica. São criados também arquivos utilizados na avaliação de critérios (arquivos descritores).

Estas informações estáticas relacionam-se ao conjunto de caminhos requeridos pelo critério *todos potenciais du-caminhos* (arquivo **pdupath.tes**); as associações requeridas pelos critérios *todos-potenciais-usos* e *todos-potenciais-usos/du* (arquivo *puassoc.tes*); os *arcos primitivos* [3] requeridos pelo critério *todos arcos* (**arcprim.tes**); e, por fim, os nós requeridos pelo critério *todos nós* (**nos_grf.tes**).

A seguir, sumariza-se os arquivos gerados pelo programa **poketool** para cada função em teste no respectivo diretório:

arcprim.tes: conjunto de arcos primitivos da função que são requeridos pelo critério "todos arcos". Os arcos primitivos são um subconjunto de todos os arcos de um programa; a execução dos arcos primitivos implica a execução dos demais arcos. São utilizados também para descrever os caminhos e associações requeridas pelos critérios potenciais usos (PU). Exemplifica-se a seguir:

```
ARCOS PRIMITIVOS DO MODULO find
  1) arco ( 2,22)
  2) arco ( 3, 4)
         .
         .
  9) arco (18,19)
 10) arco (18,20)
```

Cada linha presente no arquivo fornece um *arco primitivo* requerido pelo critério *todos arcos*, precedido pelo seu respectivo número. Neste exemplo, o arco (2,22) é o elemento requerido número 1; o arco (3,4), o número 2; e assim por diante. O número do elemento requerido é uma referência utilizada em alguns arquivos gerados pela **poketool**.

<função>.gfc: grafo textual da função referida;

grafodef.tes: descreve as variáveis definidas em cada nó do grafo da função considerada;

nos_grf.tes: contém os nós do grafo da função considerada que são requeridos pelo critério *todos nós*. Exemplo:

```
NO'S DO MODULO find
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 21  22
```

Cada número indica um nó requerido pelo critério *todos nós*. A correspondência entre cada elemento requerido e seu número respectivo é direta, ou seja, o nó 1 é o elemento requerido número 1; o nó 2, o número 2; e assim sucessivamente.

pduphath.tes: mostra os caminhos requeridos pelo critério *todos potenciais du-caminhos*. Exemplo:

```
CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS
Caminhos requeridos pelo Grafo( 1)
 1) 1 2 22
 2) 1 2 3 5 6 14 16 18 20 21 2
 3) 1 2 3 5 6 14 16 18 19 20 21 2
 4) 1 2 3 5 6 14 16 17 16
 5) 1 2 3 5 6 14 15 14
 6) 1 2 3 5 6 7 12 13 21 2
              .
              .
```

Cada linha deste arquivo representa um *potencial du-caminho* [9] (descrito por uma seqüência de nós) requerido pelo critério *todos potenciais du-caminhos* precedido pelo seu respectivo número. Para cada *Grafo(i)* [9], são mostrados todos os *potenciais du-caminhos* que iniciam no nó *i*. Neste exemplo, em relação ao *Grafo(1)*, são requeridos os potenciais du-caminhos: 1, 2, 22, de número 1; 1, 2, 3, 5, 6, 14, 16, 18, 19, 20, 1, 2, de número 2; e assim por diante.

puassoc.tes: contém as associações requeridas pelos critérios *todos potenciais usos* e *todos potenciais usos/du*.

```
ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-USOS/DU
Associacoes requeridas pelo Grafo( 1)
1) <1,(2,22),{ b, m, ns, a, n, f }>
2) <1,(3,5),{ b, m, ns, a, n, f }>
3) <1,(18,20),{ m, ns, a, n, f }>
4) <1,(18,20),{ b, m, ns, a, n, f }>
5) <1,(19,20),{ b, m, ns, n, f }>
6) <1,(18,19),{ m, ns, a, n, f }>
7) <1,(18,19),{ b, m, ns, a, n, f }>
                  .
                  .
```

Neste arquivo, cada linha representa uma associação requerida pelo critério, precedida pelo respectivo número. Para cada *Grafo(i)*, são mostradas todas as associações que iniciam no nó *i*. A descrição da associação fornece as seguintes informações, nesta ordem:

- nó no qual a(s) variável (variáveis) é (são) definida(s);

- arco do *Grafo(i)* no qual finda a associação requerida;
- variáves envolvidas na respectiva associação;

Neste exemplo, em relação ao *Grafo(1)*, são requeridas as associações:

- de número 1: associação das variáveis *b,m,ns,a,n,f* definidas no nó 1 e "potencialmente utilizáveis" no arco (2,22) sem redefinição simultânea dessas variáveis.
- de número 2: associação das variáveis *b,m,ns,a,n,f* definidas no nó 1 e "potencialmente utilizáveis" no arco (3,5) sem redefinição simultânea das variáveis.

tabvardef.tes: consiste na tabela de variáveis definidas na função considerada;

Exemplo de Utilização :

Deseja-se testar duas unidades presentes no arquivo **find.c**: a função **find** e o programa principal **main**. Para tanto aciona-se o programa **poketool** da seguinte forma:

```
poketool find.c -ffind -fmain
```

Este comando determina a análise das funções **find** e **main**, contidas no arquivo **find.c**. Caso alguma anomalia seja detectada, mensagens informativas serão geradas. Por exemplo:

- número incorreto de parâmetros;
- funções ausentes;
- descrição incorreta do diretório de *include*;
- problemas na geração da linguagem intermediária;

Em caso de sucesso, o programa reporta as seguintes mensagens ao usuário:

```
poketool - Analisador Estatico para Teste de Programas escritos
em Linguagem C
Arquivo em teste: find.c
* * Carregando Tabela de transicao lexica... * *
* * Carregando Tabela de palavras chaves... * *
* * Carregando Parser especifico da linguagem fonte... * *
* * Fazendo a analise sintatica do codigo fonte... * *
* * Nucleo POKETOOL foi bem sucedido * *

Gerando os descritores para a funcao find ...
* * Calculando os arcos primitivos... * *
* * Gerando descritores... * *
* * Geracao de Descritores da POKETOOL foi bem sucedida * *
```

```
* * Gerados os descritores da funcao find * *

Gerando os descritores para a funcao main ...
* * Calculando os arcos primitivos... * *
* * Gerando descritores... * *
* * Geracao de Descritores da POKETOOL foi bem sucedida * *
* * Gerados os descritores da funcao main * *
```

4.4 Programa Pokeexec

Objetivo: Auxiliar a execução do programa em teste para um conjunto de casos de teste.

Sinopse:

pokeexec <programa executável> [-l] [-t] [-r<num>] -f<função> [-f<função>]

Parâmetros:

<**programa executável**>: nome do programa executável a ser testado;

-f <**função**>: nome da função a ser testada. Caso se deseje testar mais de uma função do mesmo programa, deve-se informar o nome de todas.

-l: informa que o programa em teste aceita parâmetros em linha de comando;

-t: informa que o programa em teste recebe entradas pelo teclado;

-r <*num*>: informa que o teste deve iniciar do caso de teste número *num*.

Funcionamento:

É pertinente ressaltar que antes de executar o programa **pokeexec**, deve-se compilar o programa modificado para teste. Tal programa, gerado através do **poketool** utilizando o arquivo modificado para teste (testeprog.c), constitui-se no parâmetro passado ao **pokeexec**. Caso se torne necessário, deve-se preparar um *makefile* que possibilite a compilação de todos os arquivos utilizados pela unidade em teste.

Ao se acionar o programa **pokeexec**, é solicitado ao usuário entrar com as entradas via linha de comando (parâmetros do executável) e via teclado. Estas duas entradas são armazenadas em arquivos de tal forma que o controle é transferido para o programa em teste somente depois do armazenamento. Para finalizar a entrada através do teclado deve-se utilizar o caractere CTRL-D. No final da execução de cada caso de teste, o usuário pode optar por continuar o processo com outro caso de teste ou interrompê-lo.

A execução do programa modificado, através do **pokeexec**, gera informações nos subdiretórios associados às funções em teste, além de criar os subdiretórios **keyboard**, **input** e **output**.

Em cada subdiretório associado a uma função testada, são gerados arquivos que armazenam os caminhos executados pelos casos de teste daquela unidade. Considerando-se *N* como sendo o número de casos de teste executados, obtêm-se os seguintes arquivos:

path1.tes: caminho executado pelo caso de teste 1 na unidade considerada;

path2.tes: caminho executado pelo caso de teste 2 na unidade considerada;

path*N*.tes: caminho executado pelo caso de teste N na unidade considerada;

path.tes: possui o mesmo conteúdo do arquivo path*N*.tes.

O conteúdo destes arquivos tem a seguinte forma:

```
::::::::::::::
path1.tes
::::::::::::::
  1   2   3   4   6  14  15  14  15  14
 16  18  19  20  21   2   3   5   6  14
 15  14  16  17  16  17  16  18  19  20
 21   2   3   5   6  14  16  17  16  18
 20  21   2   3   5   6   7  12  13  21
  2   3   4   6  14  15  14  15  14  15
 14  15  14  16  18  19  20  21   2   3
  5   6   7   8   9  11  13  21   2  22
```

Neste exemplo está representado um caminho completo, obtido do programa testado, com a execução do caso de teste 1. Ou seja, o primeiro caso de teste provocou a execução da seqüência de nós 1,2,3,.....,21,2,22.

No subdireório **input** são criados os arquivos:

input1.tes: entrada via parâmetros utilizada no caso de teste 1;

input2.tes: entrada via parâmetros utilizada no caso de teste 2;

input*N*.tes: entrada via parâmetros utilizada no caso de teste N;

No subdireório **keyboard** são criados os arquivos:

keyboard1.tes: entrada pelo teclado utilizada no caso de teste 1;

keyboard2.tes: entrada pelo teclado utilizada no caso de teste 2;

keyboard*N*.tes: entrada pelo teclado utilizada no caso de teste N.

Tais informações poderão ser utilizadas posteriormente em *testes de regressão*.

No subdiretório **output** são gerados os arquivos:

output1.tes: saída gerada no vídeo pelo programa para o caso de teste 1;

output2.tes: saída gerada no vídeo pelo programa para o caso de teste 2;

output*N*.tes: saída gerada no vídeo pelo programa para o caso de teste N.

Exemplo de Utilização:

```
pokeexec testeprog -t -ffind -fmain -t
```

Ao se acionar o programa **pokeexec** com tais parâmetros, provoca-se e execução do programa **testeprog**. **-ffind** e **-fmain** informam o nome das funções instrumentadas pelo programa **poketool** para as quais os caminhos percorridos pelos casos de teste serão armazenados. **-t** informa que o programa recebe entrada pelo teclado. Caso recebesse também parâmetros na linha de comando, seria necessário um **-l** precedendo o **-t**.

O **pokeexec** emite as seguintes mensagens:

```
pokeexec mensagem: Comecando sessao de teste para o programa "testeprog" ...

pokeexec mensagem: removendo todos os arquivos do diretorio "output"
pokeexec: "testeprog" aceita parametros em linha de comando ? (s/n) [s]
n
pokeexec mensagem: removendo todos os arquivos do diretorio "keyboard"
pokeexec:  Executando caso de teste numero 1.
```

Neste ponto, o controle é transferido para o programa **testeprog** e tem-se a execução do primeiro caso de teste, ou seja, o programa processa os dados de entrada fornecidos e gera a respectiva saída. É emitida então a seguinte mensagem:

```
pokeexec: Voce quer continuar a executar casos de teste? (s/n)[s]
```

Caso se responda "s" à pergunta anterior, tem-se a execução do segundo caso de teste. E assim sucessivamente. Em se respondendo "n", é encerrada a execução de casos de teste.

De posse dos elementos requeridos para cada critério, gerados pelo programa **poketool**, e dos caminhos exercitados no programa em teste durante a sua execução (armazenaados pelo programa **pokeexec**), pode-se partir para a avaliação do nível de cobertura atingido pelo teste realizado. Tal tarefa é auxiliada pelo programa **pokeaval** que será descrito a seguir.

4.5 Programa Pokeaval

Objetivo: Permitir a avaliação da cobertura obtida para uma determinada função, considerando um certo conjunto de casos de teste e um critério de análise de adequação especificado.

Sinopse:

pokeaval -d<diretório da função> -pdu|-pu|-pudu|-nos|-arcs [-ne|-e] [x to y]

Parâmetros:

-d <diretório da função>: determina o diretório (criado pelo programa **poke-tool**) no qual estão as informações relativas à função cujo teste será avaliado;

-pdu|**-pu**|**-pudu**|**-nos**|**-arcs**: determina o critério a ser utilizado, para a avaliação da cobertura obtida, considerando os casos de teste aplicados.

- **-pdu**: critério todos potenciais-du-caminhos;
- **-pu**: todos potenciais-usos;
- **-pudu**: todos potenciais-usos/du;
- **-nos**: todos nós;
- **-arcs**: todos arcos;

-ne: determina a apresentação dos elementos requeridos NÃO exercitados pelos casos de teste;

-e: determina a apresentação dos elementos requeridos exercitados pelos casos de teste;

x to y: determina o escopo da avaliação como sendo do caso de teste x ao caso de teste y.

Funcionamento:

A avaliação da cobertura atingida pelo conjunto de casos de teste deve ser feita com um critério por vez. Para cada avaliação (associada a cada critério), são geradas informações no subdiretório da função analisada.

Desejando-se analisar diversas funções (unidades), deve-se executar o programa **pokeaval** para cada função a ser analisada, uma por vez.

Ao se acionar o programa passando os parâmetros corretos, são emitidas mensagens que informam sobre o processo de avaliação de cada caso de teste.

São mostrados no vídeo, como resultado da avaliação, ou os elementos requeridos pelo critério ainda NÃO cobertos, ou os elementos requeridos já efetivamente cobertos (a escolha de qual informação será fornecida é feita através da passagem do parâmetro correspondente). É fornecido também o nível de cobertura obtido (em termos percentuais).

São gerados, no subdiretório da função cujo teste foi avaliado, uma série de arquivos. Para cada critério de adequação utilizado, são gerados três arquivos distintos.

Descrevem-se agora todos os arquivos gerados no processo de avaliação para cada critério de adequação utilizado:

Critério **Todos Potenciais DU-Caminhos**:

- exec_pdu.tes;
- pduhis.tes;
- pduoutput.tes;

Critério **Todos Potenciais Usos**:

- exec_pu.tes;
- puhis.tes;

puoutput.tes;

Critério **Todos Potenciais Usos/DU**:

exec_pudu.tes;
puduhis.tes;
puduoutput.tes;

Critério **Todos Nós**:

exec_no.tes;
noshis.tes;
nosoutput.tes;

Critério **Todos Arcos**:

exec_arc.tes;
arcshis.tes;
arcoutput.tes;

Para a descrição dos mesmos, utilizar-se-a a palavra *[crit]* em substituição às siglas utilizadas pela ferramenta para referenciar os critérios. Por exemplo, a "família" de arquivos exec_*[crit]*.tes pode referenciar o critério todos nós (exec_*nos*.tes), todos potenciais usos (exec_*pu*.tes) e assim por diante.

exec_*[crit]*.tes: Informa os elementos requeridos pelo critério que foram exercitados, assim como a cobertura atingida. Por exemplo:

```
arquivo exec_pu.tes
ASSOCIACOES DO CRITERIO TODOS POT-USOS executadas:
<1,(19,20),{ b, m, ns, n, f }>
<1,(18,19),{ m, ns, a, n, f }>
<1,(18,19),{ b, m, ns, a, n, f }>
            .
            .

<19,(8,9),{ i, j, w, a }>
<19,(8,9),{ w, a }>
<20,(3,5),{ b }>

Cobertura Total = 49.375000
Media da Cobertura dos Grafo(i) = 47.688723
```

O arquivo **exec_pu.tes** acima contém todas as associações requeridas pelo critério *todos potenciais usos* que foram exercitadas durante a execução dos casos de teste. Cada linha do arquivo representa uma associação distinta. São fornecidas a média da cobertura dos *Grafos(i)* [9] e a cobertura total atingida. Note-se que a forma utilizada para descrição das associações é a mesma já pormenorizada nos arquivos gerados pelo programa **poketool**.

*[crit]***hist.tes**: contém uma lista de referências aos números dos elementos requeridos que foram exercitados. Estas referências dizem respeito ao número dos elementos requeridos nos arquivos gerados pelo programa **poketool**. Exemplo:

```
arquivo puhis.tes
5
6
7
8
9
10
11
12
13
14
15
23
28
30
.
.
```

Este arquivo contém os números das associações requeridas pelo critério *todos potenciais usos* que foram exercitadas durante a execução dos casos de teste.

*[crit]***output.tes**: Informa os elementos requeridos pelo critério que NÃO foram exercitados, assim como a cobertura atingida.

```
arquivo puoutput.tes
ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:
<1,(2,22),{ b, m, ns, a, n, f }>
<1,(3,5),{ b, m, ns, a, n, f }>
<1,(18,20),{ m, ns, a, n, f }>
            .
            .

<20,(13,21),{ b }>
<20,(8,9),{ b }>
<20,(3,4),{ b }>
Cobertura Total = 49.375000
Media da Cobertura dos Grafo(i) = 47.688723
```

O arquivo **puoutput.tes** acima contém todas as associações requeridas pelo critério *todos potenciais usos* que NÃO foram exercitadas durante a execução dos casos de teste. São fornecidas a média da cobertura dos *Grafos(i)* e a cobertura total atingida. Assim como no arquivo **exec_pu.tes**, cada linha do arquivo representa uma associação distinta.

É criado um subdiretório de nome **good***[crit]* no qual são gerados arquivos *[crit]***dif i-(i-1).tes**, sendo **i** o número do caso de teste. Nestes arquivos estão armazenadas referências ao número dos elementos requeridos que foram cobertos a mais (adicionalmente) em cada caso de teste. Note-se que estas referências dizem respeito ao número dos elementos requeridos nos arquivos gerados pelo programa **poketool**.

Ou seja, para cada diferente critério aplicado, são gerados os arquivos:

*[crit]***dif1-0.tes**: número dos elementos requeridos cobertos adicionalmente pelo caso de teste 1;

*[crit]***dif2-1.tes**: número dos elementos requeridos cobertos adicionalmente pelo caso de teste 2;

*[crit]***difN-(N-1).tes**: número dos elementos requeridos cobertos adicionalmente pelo caso de teste N (N = número de casos de teste executados).

Exemplificando, no subdiretório **/goodpu**, são gerados os arquivos:

```
::::::::::::::
pudif1-0.tes
::::::::::::::
5
6
7
12
13
14
15
23
28
.
.
::::::::::::::
pudif2-1.tes
::::::::::::::
8
9
10
11
.
.
```

Neste exemplo o arquivo **pudif1-0.tes** contém o número das associações requeridas pelo critério *todos potenciais usos* que foram cobertas pelo caso de teste 1. O arquivo **pudif2-1.tes** contém o número associações que foram adicionalmente cobertas pelo caso de teste 2, ou seja, as associaccões que foram cobertas pelo caso de teste 2 e que ainda não haviam sido cobertas anteriormente.

Vale dizer que se porventura algum caso de teste não cobrir nenhum elemento requerido adicional, o respectivo arquivo *[crit]***dif i-(i-1).tes** não será gerado. Neste caso, será emitida uma mensagem informando o fato de que o caso de teste **i** "não matou nenhuma associação".

Exemplo de Utilização:

```
pokeaval -dfind -pdu -ne 1 to 3
```

Tal comando aciona o programa **pokeaval**. **-dfind** determina o diretório da função cujo teste será avaliado (função **find**); **-pdu** determina a utilização do critério *todos* *potenciais DU-Caminhos*; **-ne** faz com que sejam mostrados os elementos requeridos ainda NÃO executados; **1 to 3** determina o escopo da avaliação, como sendo do caso de teste 1 ao caso de teste 3.

```
pokeaval - Avaliador de Casos de Teste da POKE-TOOL
pokeaval mensagem: removendo todos os arquivos do diretorio
"goodpdu"

**** Avaliando Caso de Teste Numero 1 ****

* * Realizando a avaliacao do caso de teste * *

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

**** Avaliando Caso de Teste Numero 2 ****

* * Realizando a avaliacao do caso de teste * *

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

**** Avaliando Caso de Teste Numero 3 ****

* * Realizando a avaliacao do caso de teste * *

* * Avaliacao do caso de teste foi bem sucedida * *
pokeaval: Msg: Caso de teste 3 NAO matou nenhuma associacao!


POTENCIAIS-DU-CAMINHOS que nao foram executados:

Caminhos:
1 2 22
1 2 3 5 6 14 16 18 20 21 2
1 2 3 5 6 14 16 18 19 20 21 2
1 2 3 5 6 14 16 17 16
```

Obs: listagem dos potenciais du-caminhos NÃO executados resumida para apresentação.

```
20 21 2 3 4 6 7 12 13 21
20 21 2 3 4 6 7 8 10 11 13 21
20 21 2 3 4 6 7 8 9 11 13 21

Cobertura Total = 25.384615

Media da Cobertura dos Grafo(i) = 26.717951
```

O funcionamento do **pokeaval** é análogo para todo critério aceito pelo programa.

4.6 Programa Viewgraph

Objetivo: Permitir a visualização do grafo da função em teste.

Sinopse:

viewgraph <função>.gfc

Parâmetros:

<função>.gfc: Arquivo que contém a descrição textual do grafo da função.

Funcionamento:

Ao se executar o programa **Poketool**, são geradas descrições dos grafos de todas unidades presentes no programa em teste. Tais descrições estão contidas em arquivos de nome <função>.gfc, localizados nos subdiretórios das respectivas funções. A visualização do grafo da função em teste, se relizada conjuntamente com a análise do programa modificado, permite a identificação visual dos elementos requeridos por um dado critério fornecido pela ferramenta. A **Viewgraph** [12] auxilia, portanto, de uma forma indireta, na seleção de dados de teste que sejam adequados para satisfazer o critério; além de permitir um maior entendimento acerca da estrutura de controle da função.

Os grafos gerados pela *Viewgraph* para a função utilizada no exemplo, encontram-se no **apêndice D**.

5 Apêndice A: Instalação da ferramenta

A **poketool** está disponível para *ftp anonymous* no endereço (ftp.dca.fee.unicamp.br), diretório **pub/docs/jino/poketool**.

A seguir, são descritos os passos para fazer o *download* da ferramenta.

5.1 Passos para *download* da poketool

1. Execute o comando:

   ```
   [1] ftp ftp.dca.fee.unicamp.br
   ```

   Utilize o *username anonymous* e o seu endereço de *e-mail* como *password.* Veja o exemplo abaixo:

   ```
   Name (ftp.dca.fee.unicamp.br:seu_nome):anonymous
     331 Guest login ok, send your complete e-mail address as password.
   Password:"seu_nome@nowhere.com"
   ```

2. No *prompt* do *ftp*, deve-se executar a seguinte seqüência de comandos:

   ```
   ftp> cd pub/docs/jino/poketool
   ftp> bin
   ftp> mget *
   ```

   Depois de concluída a transferência, deve-se executar:

   ```
   ftp>bye
   ```

5.2 Instalação da poketool

Para instalar a **poketool**, devem ser executados os passos a seguir. Recomenda-se que os comandos sejam executados no diretório de instalação definitivo da ferramenta (e.g. <Diretório de Ferramentas>).

1. Descomprimir o arquivo **poke.tar.Z**:

   ```
   [2] uncompress poke.tar.Z
   ```

2. Separar os arquivos através do utilitário **tar**:

   ```
   [3] tar -xvf poke.tar
   ```

   São gerados os seguintes diretórios a partir do corrente:

   ```
   <Diretorio de Ferramentas>/poketool/bin
   <Diretorio de Ferramentas>/poketool/bin/litabs
   <Diretorio de Ferramentas>/poketool/bin/poketabs
   ```

   Os executáveis estão no diretorio `poketool/bin` e as tabelas utilizadas, respectivamente, nos diretorios `poketool/bin/litabs` e `poketool/bin/poketabs`.

3. Alterar o arquivo *.cshrc* ou equivalente dos futuros usuários da **poketool** incluindo os seguintes comandos (ou equivalentes):

   ```
   set path = ($path <Diretorio de Ferramentas>/poketool/bin)
   setenv POKEBIN  <Diretorio de Ferramentas>/poketool/bin
   setenv LITABS   <Diretorio de Ferramentas>/poketool/bin/litabs
   setenv POKETABS <Diretorio de Ferramentas>/poketool/bin/poketabs
   ```

Está concluída a instalação.

6 Apêndice B: Listagem do programa de exemplo

```
/*--------------------------------------------------
 *
 *   Program Name:      Find ( C version)
 *
 *
 *--------------------------------------------------
 */

int a [256+1];

void find(int n,int f)
{
    int  b;
    int   m, ns, i, j, w;

    b = 0;
    m = 1;
    ns = n;

   while ((m < ns)  || b)
   {   if   (!b)
       {  i = m;
          j = ns;
       }
       else
          b = 0;
   /*-------------------------------*/
       if   (i > j)
       {    if   (f > j)
                 {  if  (i > f)
                        m = ns;
                    else
                        m = i;
                 }
            else
                 ns = j;
       }
   /*-------------------------------*/
       else
       {    while  (a[i] < a[f])
                   i = i + 1 ;
            while  (a[f] < a[j])
                   j = j - 1 ;
            if   (i <= j)
            {    w = a[i];
 a[i] = a[j];
                 a[j] = w;
```

```
                i = i + 1;
                j = j - 1;
            }
            b = 1;
        }
    }
}

/*------------------------------------------------------------------
 *
 *   Program Name:      main function of Find1 ( C version)
 *
 *   Description:
 *
 *      INPUT:  an array a and an index f
 *
 *      OUTPUT: permutes the elements of a so that all elements
 *              to the left of position f are less than or equal
 *              to a[f] and all elements to the right of position
 *              f are greater than or equal to a[f].
 *
 *      ORIGIN: C. Hoare.  Proof of a program: Find.
 *              Commun. ACM, 14(1):39, 1971
 *
 *      OUR VERSION: This C version is translated from the
 *                   Pascal version used by Phyllis Frankl.
 *
 *   ATTENTION:
 *      The index of a C array ........
 *
 *
 *   BUG:
 *      This program gives an infinite loop if the value of
 *      n leass than the value of f.  This is because the while
 *      loop a[i] < a[f] inside the function "find".
 *      (a[f] is not initialized)
 *
 *
 *      Example:
 *      >>>
 *      >>> cat btest31c.3
 *      >>> 3      [ n = 3 ]
 *      >>> 8      [ f = 8 ]
 *      >>> 100    [ a[1]  ]   [ We skip a[0] in C ]
 *      >>> 30     [ a[2]  ]
 *      >>> 150    [ a[3]  ]
 *      >>>
 *      >>> a.out <btest31c.3
 *          \
```

```
 *          \
 *          \------  an infinite loop
 *
 *
 *------------------------------------------------------------------
 */


/* int      a[max + 1];   */ /*=====================================================*/
                         /*  Because we skip index = 0, we need to declare   */
                         /*  "a" as an integer  array with MAX + 1 elements. */
                         /*                                                  */
                         /*  Ref: p249 C by Dissection                       */
                         /*==================================================*/

main()
{  char  *mystr;
   int  i;
   int N, F;

   char  *gets();
   void  find();

   mystr = (char *) malloc (80);  /*--------------------------------
                                   * allocate space to mystr
                                   *--------------------------------
                                   */

   scanf("\%d", &N);
   gets(mystr);         /*------------------------------------------
                         *  We use gets() to mimic readln() in the
                         *  original Pascal program.
                         *  gets() reads a string into s from stdin
                         *  until a new-line character is read.
                         *------------------------------------------
                         */
   scanf("\%d", &F);
   gets(mystr);

   for (i=1;i<=N;i++)
   {
      scanf("\%d", &a[i]);
      gets(mystr);
   }


   find(N, F);
   printf("\%5d\n", N);
   printf("\%5d\n", F);
```

```
   for (i=1; i<=N; i++)
      printf("\%5d\n", a[i]);
}
```

7 Apêndice C: Arquivos Gerados pela poketool

Um programa de ordenação conhecido como *buble sort* foi testado com o auxílio da **poketool**. Mostram-se neste apêndice a listagem do programa e os arquivos gerados em cada etapa do teste. Não serão apresentados arquivos intermediários utilizados pela ferramenta, mas somente aqueles que, de alguma forma, contribuem para um maior entendimento acerca do processo de teste por parte do usuário.

```
arquivo bbsort.c

#include <stdio.h>

int A[30], N;

main(){
int j, i;
  printf("Number of elements = ");
  scanf("%d\n",&N);
  for(i=0; i<N; i++){
    scanf("%d", &A[i]);
    printf("\n");
  }
  sort();
  for(i=0; i<N; i++){
    printf("A[%d]= %d\n", i, A[i]);
  }
}


sort()
{
  int pos=0, i1=0, i2, high;
  while (i1<N-1){  /* Scan array */
    high = A[i1];
    i2 = i1+1;
    pos = i1;
    while (i2<N){      /* Largest */
      if (A[i2]>high){
        high=A[i2];
        pos=i2;
      }
```

```
      i2=i2+1;
    }                    /* Exchange */
    i2=A[i1];
    A[i1]=high;
    A[pos]=i2;
    i1=i1+1;
  }
}
```

Aciona-se o comando:

```
poketool bbsort.c -fsort
```

7.1 Arquivos gerados pelo programa poketool

```
arquivo testeprog.c

#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>


#include <stdio.h>
int A[30], N;

main()
{
   int j, i;
   printf("Number of elements = ");
   scanf("%d\n",&N);
   for(i=0;i<N;i++)
      {
      scanf("%d", &A[i]);
      printf("\n");
      }
   sort();
   for(i=0;i<N;i++)
      {
      printf("A[%d]= %d\n", i, A[i]);
      }
}

sort()
/*  1 */        {
                   FILE * path = fopen("sort/path.tes","a");
                   static int printed_nodes = 0;
/*  1 */           int pos=0, i1=0, i2, high;
```

```
                    ponta_de_prova(1);
/*  2 */            while(i1<N-1)
/*  3 */            {
                       ponta_de_prova(2);
                       ponta_de_prova(3);
/*  3 */               high = A[i1];
/*  3 */               i2 = i1+1;
                       pos = i1;
/*  4 */               while(i2<N)
/*  5 */               {
                          ponta_de_prova(4);
                          ponta_de_prova(5);
/*  5 */                  if(A[i2]>high)
/*  6 */                  {
                             ponta_de_prova(6);
/*  6 */                     high=A[i2];
/*  6 */                     pos=i2;
/*  6 */                  }
                          ponta_de_prova(7);
/*  7 */                  i2=i2+1;
/*  7 */               }
                       ponta_de_prova(4);
                       ponta_de_prova(8);
/*  8 */               i2=A[i1];
/*  8 */               A[i1]=high;
/*  8 */               A[pos]=i2;
/*  8 */               i1=i1+1;
/*  8 */            }
                    ponta_de_prova(2);
                    ponta_de_prova(9);
                    fclose(path);
/*  9 */         }
```

No subdiretório **/sort**, criado pelo **poketool**, foram gerados os seguintes arquivos:

```
arquivo  arcprim.tes

ARCOS PRIMITIVOS DO MODULO sort
  1) arco ( 2, 9)
  2) arco ( 4, 8)
  3) arco ( 5, 6)
  4) arco ( 5, 7)


arquivo nos_grf.tes

NO'S DO MODULO sort
  1   2   3   4   5   6   7   8   9
```

```
arquivo  pdupaths.tes

CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS

Caminhos requeridos pelo Grafo( 1)
 1) 1 2 9
 2) 1 2 3 4 8 2
 3) 1 2 3 4 5 7 4
 4) 1 2 3 4 5 6 7 4

Caminhos requeridos pelo Grafo( 3)
 5) 3 4 8 2 9
 6) 3 4 8 2 3
 7) 3 4 5 7 4
 8) 3 4 5 6 7

Caminhos requeridos pelo Grafo( 6)
 9) 6 7 4 8 2 9
10) 6 7 4 8 2 3
11) 6 7 4 5 7
12) 6 7 4 5 6

Caminhos requeridos pelo Grafo( 7)
13) 7 4 8
14) 7 4 5 7
15) 7 4 5 6 7

Caminhos requeridos pelo Grafo( 8)
16) 8 2 9
17) 8 2 3 4 8
18) 8 2 3 4 5 7 4
19) 8 2 3 4 5 6 7 4
```

```
arquivo  puassoc.tes

ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-USOS/DU
Associacoes requeridas pelo Grafo( 1)
1) <1,(2,9),{ pos, i1, A, N }>
2) <1,(8,2),{ N }>
3) <1,(4,8),{ i1, A, N }>
4) <1,(5,7),{ i1, A, N }>
5) <1,(7,4),{ i1, A, N }>
6) <1,(5,6),{ i1, A, N }>
7) <1,(2,3),{ pos, i1, A, N }>

Associacoes requeridas pelo Grafo( 3)
8) <3,(2,9),{ pos, high }>
```

```
9) <3,(2,3),{ pos, high }>
10) <3,(4,8),{ pos, i2, high }>
11) <3,(7,4),{ pos, high }>
12) <3,(5,7),{ pos, i2, high }>
13) <3,(6,7),{ i2 }>
14) <3,(5,6),{ pos, i2, high }>

Associacoes requeridas pelo Grafo( 6)
15) <6,(2,9),{ pos, high }>
16) <6,(2,3),{ pos, high }>
17) <6,(4,8),{ pos, high }>
18) <6,(5,7),{ pos, high }>
19) <6,(5,6),{ pos, high }>

Associacoes requeridas pelo Grafo( 7)
20) <7,(4,8),{ i2 }>
21) <7,(5,7),{ i2 }>
22) <7,(6,7),{ i2 }>
23) <7,(5,6),{ i2 }>

Associacoes requeridas pelo Grafo( 8)
24) <8,(2,9),{ i1, i2, A }>
25) <8,(4,8),{ i1, A }>
26) <8,(5,7),{ i1, A }>
27) <8,(7,4),{ i1, A }>
28) <8,(5,6),{ i1, A }>
29) <8,(2,3),{ i1, i2, A }>
```

Neste arquivo, cada elemento requerido pelo critério, é representado por uma linha que descreve, nesta ordem:

- nó no qual a(s) variável (variáveis) é (são) definida(s);

- arco no qual finda a associação requerida;

- variáves envolvidas na respectiva associação;

Compila-se então o programa instrumentado gerado pelo **poketool** (testeprog.c), obtendo-se a versão executável do mesmo (de nome testeprog no exemplo considerado).

```
gcc testeprog.c -o testeprog
```

Inicia-se agora a fase dinâmica do teste. Executaram-se três casos de teste através do programa **pokeexec** . Serão relatados os arquivos criados.

7.2 Arquivos gerados pelo programa pokeexec

Arquivos gerados no subdiretório /keyboard:

```
::::::::::::::
keyboard1.tes
::::::::::::::
5
95
80
75
70
65
::::::::::::::
keyboard2.tes
::::::::::::::
4
10
20
30
40
::::::::::::::
keyboard3.tes
::::::::::::::
3
26
26
26
```

Arquivos gerados no subdiretório /output:

```
::::::::::::::
output1.tes
::::::::::::::
A[0]= 95
A[1]= 80
A[2]= 75
A[3]= 70
A[4]= 65

::::::::::::::
output2.tes
::::::::::::::
A[0]= 40
A[1]= 30
A[2]= 20
A[3]= 10

::::::::::::::
```

```
output3.tes
::::::::::::::
A[0]= 26
A[1]= 26
A[2]= 26
```

Arquivos gerados no subdiretório /sort:

```
::::::::::::::
path.tes
::::::::::::::
  1    2    3    4    5    6    7    4    5    7
  4    5    6    7    4    5    7    4    5    7
  4    5    7    4    5    7    4    5    7    4
  5    7    4    5    7    4    5    7    4    5
```

Obs: Listagem resumida para apresentação.

```
  7    4    5    6    7    4    8    2    3    4
  5    6    7    4    5    7    4    8    2    3
  4    5    7    4    8    2    9    1    2    3    4    5    7    4    5    7    4
  8    2    3    4    5    7    4    8    2    9
::::::::::::::
path1.tes
::::::::::::::
  1    2    3    4    5    6    7    4    5    7
  4    5    6    7    4    5    7    4    5    7
  4    5    7    4    5    7    4    5    7    4
  5    7    4    5    7    4    5    7    4    5
```

Obs: Listagem resumida para apresentação.

```
  5    7    4    5    7    4    8    2    3    4
  5    7    4    5    7    4    5    7    4    8
  2    3    4    5    7    4    5    7    4    8
  2    3    4    5    7    4    8    2    9
::::::::::::::
path2.tes
::::::::::::::
  1    2    3    4    5    6    7    4    5    7
  4    5    6    7    4    5    7    4    5    7
  4    5    7    4    5    7    4    5    7    4
  5    7    4    5    7    4    5    7    4    5
```

Obs: Listagem resumida para apresentação.

```
  2    3    4    5    7    4    8    2    9    1    2    3    4    5    6    7    4    5    6
  7    4    5    6    7    4    8    2    3    4
  5    6    7    4    5    7    4    8    2    3
  4    5    7    4    8    2    9
::::::::::::::
```

```
path3.tes
::::::::::::::
  1    2    3    4    5    6    7    4    5    7
  4    5    6    7    4    5    7    4    5    7
  4    5    7    4    5    7    4    5    7    4
  5    7    4    5    7    4    5    7    4    5
```

Obs: Listagem resumida para apresentação.

```
  7    4    5    6    7    4    8    2    3    4
  5    6    7    4    5    7    4    8    2    3
  4    5    7    4    8    2    9    1    2    3    4    5    7    4    5    7    4
  8    2    3    4    5    7    4    8    2    9
```

7.3 Arquivos gerados pelo programa pokeaval

Após a execução dos casos de teste (três casos no exemplo em questão), pode-se avaliar o nível de cobertura atingido pelo teste, para um dado critério, utilizando-se o programa **pokeaval**. Nesta fase são gerados os seguintes arquivos:

Avaliação segundo o critério todos os nós

```
arquivo exec_no.tes

NOS DO CRITERIO TODOS-NOS executados:

  1    2    3    4    5    6    7    8    9

Cobertura Total = 100.000000



arquivo nosoutput.tes

NOS DO CRITERIO TODOS-NOS nao executados:


Cobertura Total = 100.000000
```

No subdiretório /goodnos:

```
arquivo nosdif1-0.tes
1
2
3
4
5
6
```

```
7
8
9
```

Avaliação segundo o critério todos os arcos:

```
arquivo exec_arc.tes

ARCOS DO CRITERIO TODOS-ARCOS executados:

arco ( 2, 9)
arco ( 4, 8)
arco ( 5, 6)
arco ( 5, 7)

Cobertura Total = 100.000000


arquivo arcoutput.tes

ARCOS DO CRITERIO TODOS-ARCOS nao executados:


Cobertura Total = 100.000000
```

No subdiretório /goodarcs :

```
arquivo arcsdif1-0.tes
1
2
3
4
```

Avaliação segundo o critério Todos Potenciais Usos:

```
arquivo exec_pu.tes

ASSOCIACOES DO CRITERIO TODOS POT-USOS executadas:

<1,(8,2),{ N }>
<1,(4,8),{ i1, A, N }>
<1,(5,7),{ i1, A, N }>
<1,(7,4),{ i1, A, N }>
<1,(5,6),{ i1, A, N }>
<1,(2,3),{ pos, i1, A, N }>
```

```
<3,(2,9),{ pos, high }>
<3,(2,3),{ pos, high }>
<3,(7,4),{ pos, high }>
<3,(5,7),{ pos, i2, high }>
<3,(6,7),{ i2 }>
<3,(5,6),{ pos, i2, high }>
<6,(2,3),{ pos, high }>
<6,(4,8),{ pos, high }>
<6,(5,7),{ pos, high }>
<6,(5,6),{ pos, high }>
<7,(4,8),{ i2 }>
<7,(5,7),{ i2 }>
<7,(6,7),{ i2 }>
<7,(5,6),{ i2 }>
<8,(2,9),{ i1, i2, A }>
<8,(4,8),{ i1, A }>
<8,(5,7),{ i1, A }>
<8,(7,4),{ i1, A }>
<8,(5,6),{ i1, A }>
<8,(2,3),{ i1, i2, A }>

Cobertura Total = 89.655172
Media da Cobertura dos Grafo(i) = 90.285721


arquivo puoutput.tes

ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

<1,(2,9),{ pos, i1, A, N }>
<3,(4,8),{ pos, i2, high }>
<6,(2,9),{ pos, high }>

Cobertura Total = 89.655172
Media da Cobertura dos Grafo(i) = 90.285721
```

No subdiretório /goodpu:

```
::::::::::::::
pudif1-0.tes
::::::::::::::
2
3
4
5
6
7
8
```

```
9
11
12
13
14
16
17
18
19
20
21
22
23
24
25
26
27
28
29
```

Avaliação segundo o critério Todos Potenciais Usos/DU:

```
arquivo exec_pudu.tes

ASSOCIACOES DO CRITERIO TODOS POT-USOS/DU executadas:
<1,(5,7),{ i1, A, N }>
<1,(7,4),{ i1, A, N }>
<1,(5,6),{ i1, A, N }>
<1,(2,3),{ pos, i1, A, N }>
<3,(7,4),{ pos, high }>
<3,(5,7),{ pos, i2, high }>
<3,(6,7),{ i2 }>
<3,(5,6),{ pos, i2, high }>
<6,(2,3),{ pos, high }>
<6,(4,8),{ pos, high }>
<6,(5,7),{ pos, high }>
<6,(5,6),{ pos, high }>
<7,(4,8),{ i2 }>
<7,(5,7),{ i2 }>
<7,(6,7),{ i2 }>
<7,(5,6),{ i2 }>
<8,(2,9),{ i1, i2, A }>
<8,(5,7),{ i1, A }>
<8,(7,4),{ i1, A }>
<8,(5,6),{ i1, A }>
<8,(2,3),{ i1, i2, A }>
```

```
Cobertura Total = 72.413793
Media da Cobertura dos Grafo(i) = 75.523810

arquivo puduoutput.tes

ASSOCIACOES DO CRITERIO TODOS POT-USOS/DU nao executadas:
<1,(2,9),{ pos, i1, A, N }>
<1,(8,2),{ N }>
<1,(4,8),{ i1, A, N }>
<3,(2,9),{ pos, high }>
<3,(2,3),{ pos, high }>
<3,(4,8),{ pos, i2, high }>
<6,(2,9),{ pos, high }>
<8,(4,8),{ i1, A }>

Cobertura Total = 72.413793
Media da Cobertura dos Grafo(i) = 75.523810
```

No subdiretório /goodpudu:

```
::::::::::::::
pududif1-0.tes
::::::::::::::
4
5
6
7
11
12
13
14
16
17
18
19
20
21
22
23
24
26
27
28
29
```

Avaliação segundo o critério Todos Potenciais DU-Caminhos :

```
arquivo exec_pdu.tes

POTENCIAIS-DU-CAMINHOS que foram executados:
Caminhos:
1 2 3 4 5 7 4
1 2 3 4 5 6 7 4
3 4 5 7 4
3 4 5 6 7
6 7 4 8 2 3
6 7 4 5 7
6 7 4 5 6
7 4 8
7 4 5 7
7 4 5 6 7
8 2 9
8 2 3 4 5 7 4
8 2 3 4 5 6 7 4

Cobertura Total = 68.421053
Media da Cobertura dos Grafo(i) = 70.000000


arquivo pduoutput.tes

POTENCIAIS-DU-CAMINHOS que nao foram executados:
Caminhos:
1 2 9
1 2 3 4 8 2
3 4 8 2 9
3 4 8 2 3
6 7 4 8 2 9
8 2 3 4 8

Cobertura Total = 68.421053
Media da Cobertura dos Grafo(i) = 70.000000
```

No subdiretório */goodpdu*:

```
::::::::::::::
pudif1-0.tes
::::::::::::::
3
4
7
8
10
11
```

```
12
13
14
15
16
18
19
```

8 Apêndice D: Grafo da função find

Apresenta-se a seguir o grafo da função **find**, gerado com o auxílio do programa **Viewgraph**.

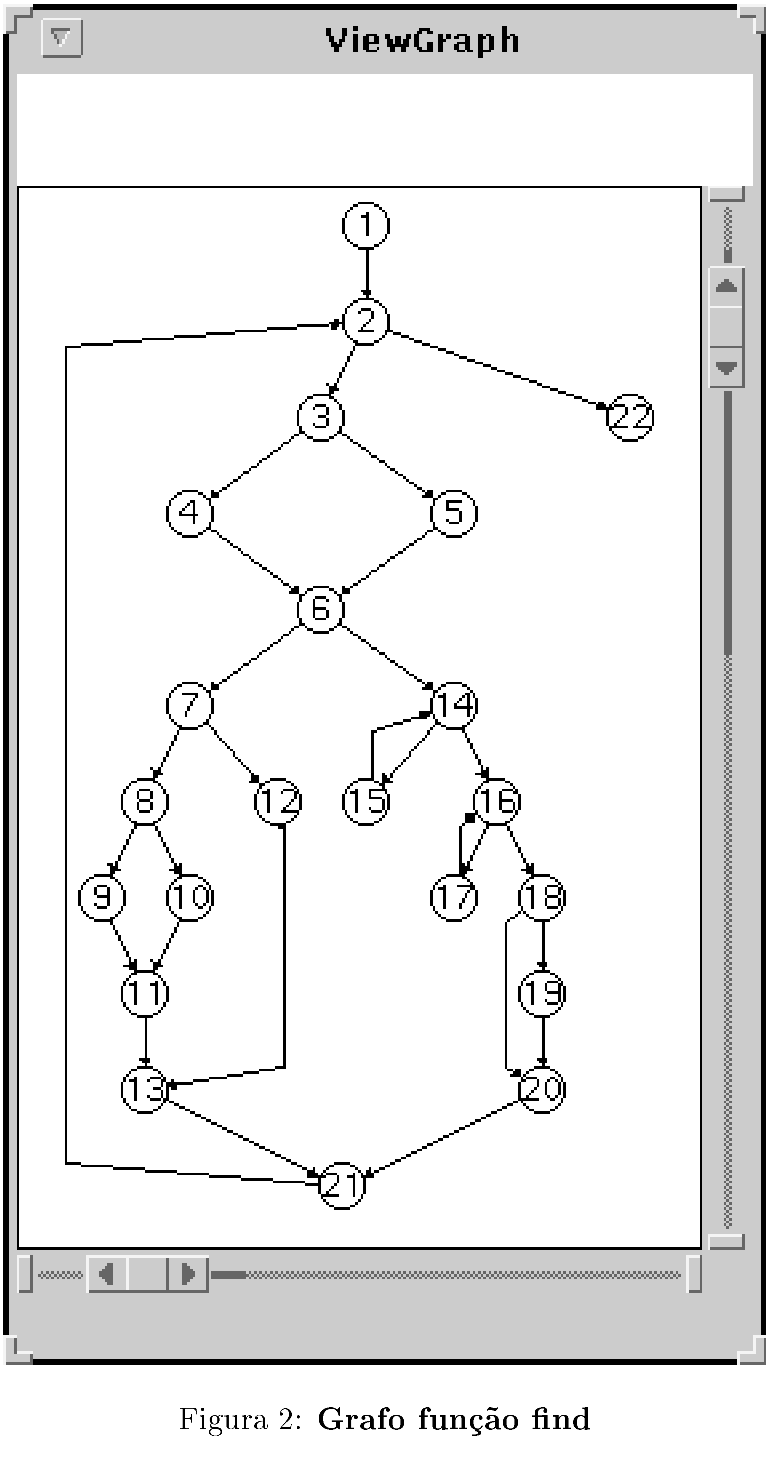

Figura 2: **Grafo função find**

Referências

[1] M. Carnassale *Tese de Mestrado*, DCA/FEE/UNICAMP Fev. , 1991.

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

[3] Chusho, T. "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing," *IEEE Trans. Software Eng.* Vol. SE - 13, No.5, Maio 1987, pp.509-517.

[4] F. G. Frankl e E.J. Weyuker "An Aplicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Eng.* Vol. 14, No. 10, Oct. 1988, pp. 1483-1498.

[5] Herman, P. M. "Data Flow Approach to Program Testing," *Australian Computer Journal*, Vol. 8, No.3, Nov. 1976.

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

[7] Laski, J.W. e Korel, B. "A Data Flow Oriented Program Testing Strategy", IEEE Trans. on Software Eng., 9(3), 347-354, May, 1983.

[8] Maldonado, J.C; Chaim, M.L. e Jino, M. "Seleção de Casos de Teste Baseada nos Critérios Potenciais Usos", II Simpósio Brasileiro de Engenharia de Software, Canela, RS, Out., 1988.

[9] Maldonado, J.C. Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. *Tese de Doutorado*, DCA/FEE/ UNICAMP - Campinas, SP, Julho, 1991.

[10] S. C. Ntafos "On Required Element Testing," *IEEE Trans. Software Eng.*, Vol. SE - 10, pp. 795-803, Nov. 1984.

[11] Vergilio, S.R. Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas. *Tese de Mestrado*, DCA/FEE/UNICAMP, Campinas, SP, Jan., 1992.

[12] Vilela, P.R. Uma Ferramenta para Auxilio Visual ao Teste e depuração de programas. *Tese de Mestrado*,DCA/FEE/UNICAMP, Campinas, SP, Mar., 1994.