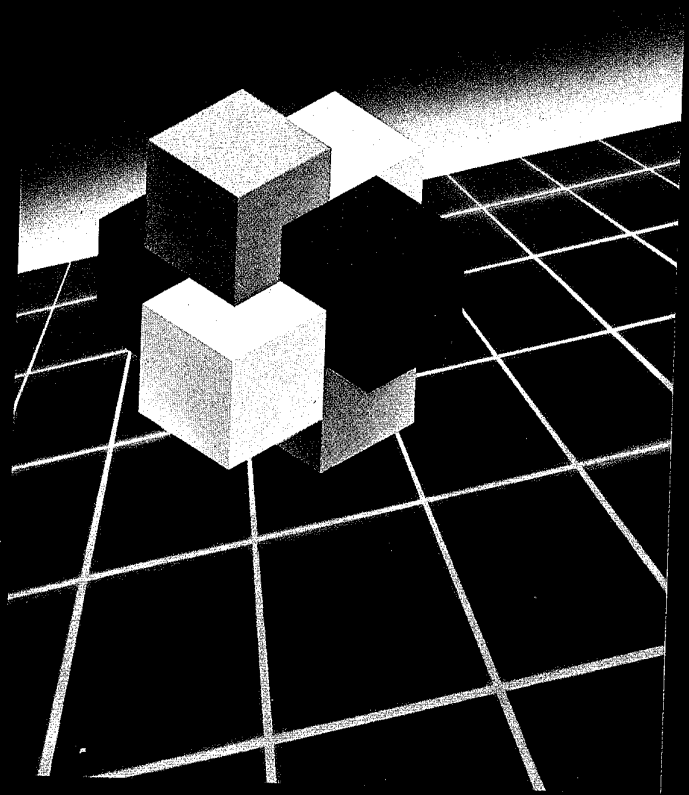
PANEL '85
EXPODATA

20 a 27 de Julho de 1985
UFRGS - Porto Alegre - Brasil

# V Congresso da Sociedade Brasileira de Computação
# XI Conferência Latino-Americana de Informática



004.06
S678
1985
V.1

ANAIS vol. I

# UM CRITERIO DE SELEÇAO DE CASOS TESTE POR COBERTURA

ARNDT VON STAA*

SUMARIO

E descrito um critério para a seleção de casos teste por cobertura. O critério parte de diagramas de organização modular do programa (diagramas estruturados), criando uma expressão regular descrevendo todos os caminhos existentes neste programa. De posse desta expressão regular são determinados os caminhos a serem executados, um para cada caso teste. Os caminhos são selecionados utilizando-se um critério que procura simular indução matemática para o caso de repetições. Ao final são discutidas extensões e algumas limitações do critério.

ABSTRACT

An open box test data selection criterion is described. The criterion departs from a graphical representation of the program's structure. A regular expression is derived from this structure, describing all paths of the program to be tested. Once the regular expression has been formed, the paths to be tested are identified. The identification process attempts to simulate mathematical induction.

* PhD em Ciência da Computação. Areas de interesse: engenharia de software, ambientes de desenvolvimento de software, controle de qualidade de software. E professor associado no Departamento de Infomática da Pontifícia Universidade Católica do Rio de Janeiro

Este artigo recebeu apoio de: FINEP contrato: 62.84.0416.00 e de FIPEC contrato: 1468-0

Teste de programas continua a ser um problema relevante, apesar dos resultados já alcançados na área de prova formal de programas. Testar um programa não é submetê-lo a uns tantos testes e "ver no que dá". Testar é, antes de tudo, a seleção racional de experimentos (casos testes) e a condução controlada destes experimentos (testes) [Myers79]. O teste sistemático de um programa visa identificar as deficiências deste programa. Caso não identifique deficiências, o teste sistemático deve ser capaz de atestar o grau de confiabilidade do programa. É claro que, no caso geral, testes não conseguirão assegurar a correção absoluta de um programa. Tampouco a prova formal tem-se mostrado capaz deste intento [Gerhard77]. Admitem os menos radicais que a virtude está a meio caminho. Ou seja, deve-se balancear o esforço investido em testes e em prova de correção. O resultado disto será um programa confiável desenvolvido com economia de recursos [Howden80].

Um dos critérios de seleção de casos teste é a cobertura de caminhos [Beizer83]. Este critério de cobertura de caminhos gera, a partir da organização interna do programa, casos teste que cubram todos os caminhos de um determinado conjunto de caminhos definidos no programa. É amplamente conhecido que o conjunto de todos os caminhos de determinado programa pode ser infinito. Precisa-se, pois, selecionar casos teste de modo que:

i- o conjunto de caminhos a serem examinados seja finito e relativamente pequeno, de modo que o custo do teste não exceda o necessário para assegurar o alcance do nível de qualidade desejado

ii- o conjunto de caminhos a serem examinados seja uma seleção válida. Uma seleção é válida [Goodenough77] se, existindo falha no programa, será gerado pelo menos um caso teste que exercite esta falha e acuse um erro, i.e. uma diferença entre o resultado esperado e o resultado computado.

Critérios de seleção de casos teste caixa aberta, em particular a cobertura de caminhos, têm-se baseado no fluxograma do programa. Na inexistência de geradores automáticos de casos teste, isto é no mínimo um inconveniente, uma vez que fluxogramas tendem a gerar somente trabalho adicional, pouco ou nada contribuindo para um melhor entendimento do programa. Em outras palavras, um bom critério de seleção de casos teste deve auxiliar o controlador de qualidade a esmiuçar o programa, auxiliando-o e motivando-o a argumentar formalmente a correção do programa sendo examinado.

O critério de seleção apresentado neste artigo parte de diagramas de organização modular de programas. Estes diagramas são o resultado do projeto do programa. Ou seja, a base para a seleção de casos teste é o projeto do programa e não o seu fluxograma. O primeiro passo é converter diagramas de organização modular em expressões algébricas (seção 2). Para programas não recursivos, a expressão algébrica é uma expressão regular. A linguagem regular correspondente a esta expressão algébrica é o conjuntó de todos os caminhos possíveis no programa. De posse da expressão algébrica é realizada uma seleção criteriosa visando determinar o conjunto de expressões de caminhos a serem percorridos pelos casos teste (seção 3). Cada um destes caminho corresponde a um caso teste. Ao criar caminhos envolvendo ciclos procura-se simular a indução matemática como um meio de tornar finito o número de caminhos a testar. Finalmente, na seção 4 são examinados alguns problemas pendentes, e é feita uma avaliação do critério de seleção.

## 2. Geração da expressão algébrica

Como já foi mencionado, o ponto de partida do critério de seleção de casos teste é o diagrama de estrutura organizacional do programa (diagrama estruturado). Adotamos aqui uma linguagem de representação semelhante á descrita em [Staa83]. A esta linguagem de representação são adicionadas as regras:

i- cada bloco de ativação repetitiva contém uma única repetição. Esta ativa um bloco de inicialização e um único bloco executor das instâncias desta repetição

ii- cada bloco de ativação seletiva contém uma única seleção

A figura 1 ilustra resumidamente a linguagem de representação gráfica utilizada neste artigo. A figura 2 ilustra o uso desta linguagem de representação em um projeto detalhado de programa.

Cabe observar que a presente linguagem de representação é capaz de representar qualquer programa procedural. Além disso é fácil mostrar que as linguagens de representação utilizadas por [Yourdon79] e por [Jackson75] são conversíveis para a linguagem de representação adotada neste artigo.

Para efeito deste artigo impomos as seguintes restrições adicionais:

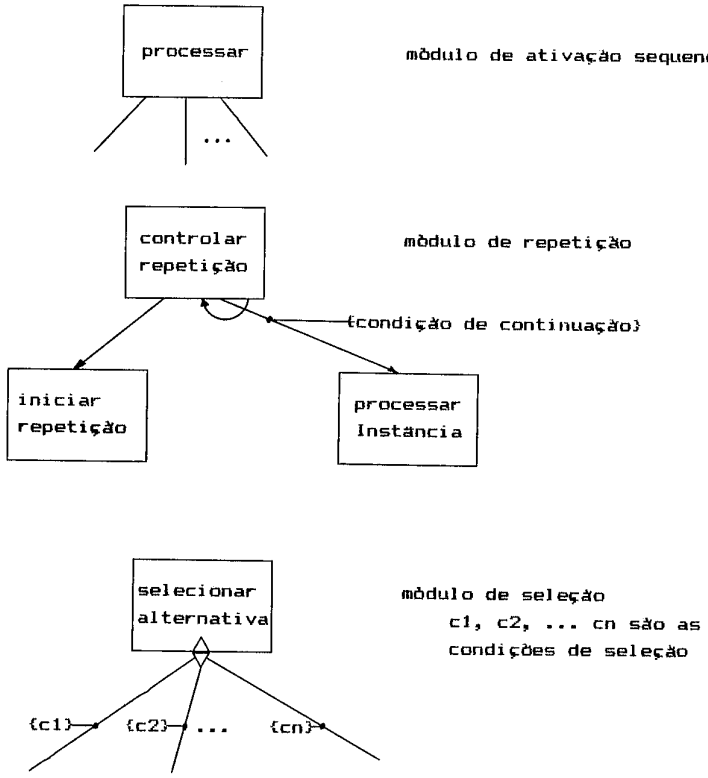



Figura 1. Descrição da linguagem de representação

i- não é permitida a recursão. Ou seja, o grafo representativo do projeto do programa é acíclico. No entanto, não é necessário que seja puramente arborescente (i.e. pode ter "fan in" maior do que 1).

ii- não é permitida a suspensão da execução (aborto) em módulos quaisquer. Ou seja, entrada e saída são sempre via o módulo raiz do programa.

Estas restrições não são demasiadamente fortes. Primeiro, uma grande gama de programas interessantes é formada por programas não recursivos. Na seção 4 será comentado um possível caminho para eliminar

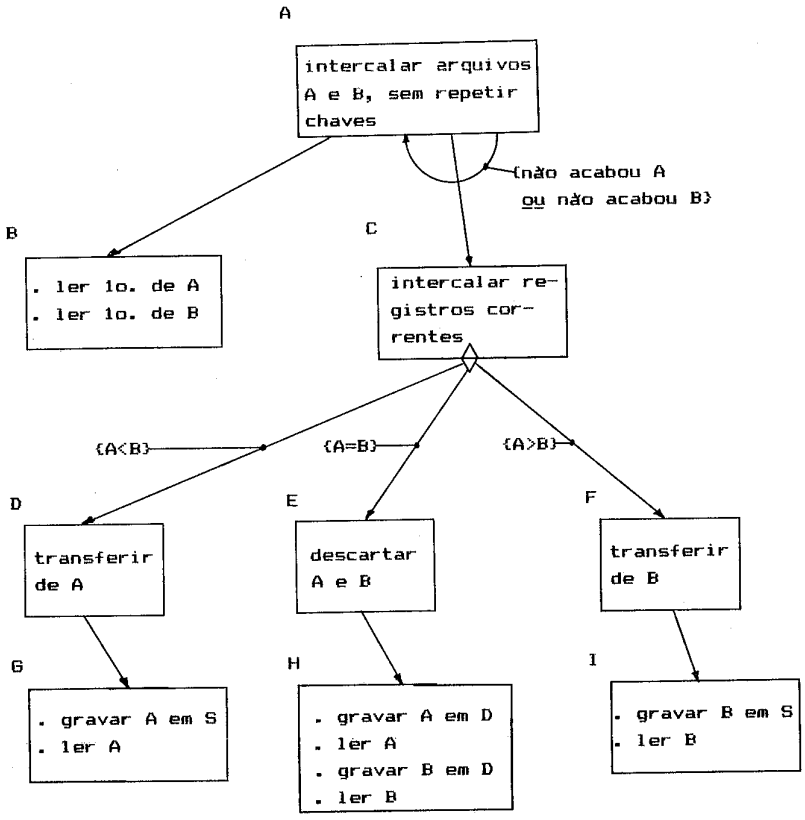



Figura 2. Exemplo de projeto detalhado. Programa de intercalação de arquivos sem repetir chave. O arquivo resultado è S. O arquivo de erros - chaves duplicadas - è D. Entradas são arquivos A e B, assumidos corretos.

esta restrição. E, segundo, a interdição de supensão em qualquer lugar pode ser resolvida através do uso de um sinal indicador de condição de continuação. Esta regra è equivalente á regra bàsica da programação estruturada, ou seja, cada bloco tem uma ùnica entrada e uma ùnica saida.

Antes de iniciar a geração da expressão algèbrica do grafo

correspondente á estrutura modular do programa, rotula-se este grafo. No exemplo da figura 2 utilizamos letras para rotular os blocos. Após executa-se o algoritmo descrito na figura 3. A figura 4 ilustra alguns passos da execução deste algoritmo utilizando o programa da figura 2.

```
gerar expressão( raiz )

   caso tipo do bloco raiz

      raiz nula:
         faz nada

      bloco simples:
         imprimir rótulo do bloco raiz
         para  cada aresta do bloco raiz  repetir
            gerar expressão( bloco ativado por[ aresta ])
         fim repetir

      bloco condicional:
         imprimir rótulo do bloco raiz
         imprimir "("
         gerar expressão( bloco ativado por[ primeira aresta])
         para cada aresta de bloco raiz,
                     a partir da segunda inclusive repetir
            imprimir "¦"
            gerar expressão( bloco ativado por[ aresta ])
         fim repetir
         se condição é opcional
         então
             imprimir "¦"
             imprimir "ø"          /* nulo */
         fim se
         imprimir ")"

      bloco repetitivo:
         imprimir rótulo do bloco raiz
         gerar expressão( inicializar raiz)
         imprimir "["
         gerar expressão( repetir raiz )
         imprimir "]*"

   fim caso
fim gerar expressão
```

Figura 3. Algoritmo recursivo para a geração da expressão algèbrica

ao chegar em "intercalar registro"

A B [ C

ao terminar o algoritmo

A B [ C ( D G ¦ E H ¦ F I ) ]*

após incorporar o arrasto (seção 3)

A B [2 C ( D G ¦ E H ¦ F I ) ]*

Figura 4. Ilustração da execução do algoritmo

## 3. Geração de expressões de caminho

Uma vez de posse da expressão algébrica que descreve todos os caminhos existentes no programa sendo examinado, passa-se a selecionar o conjunto de caminhos que deverão ser percorridos durante os testes.

A primeira providência a ser tomada é determinar um critério para a escolha de caminhos envolvendo repetições, uma vez que estas tendem a tornar infinito o número de caminhos a serem examinados.

Repetições dependem de uma variável ou de um estado de controle (exemplo de estado de controle: registro lido). Em adição variáveis e/ou estados são modificados a cada ciclo. Em princípio pode-se imaginar a existência de funções, uma para cada variável ou estado modificado durante o ciclo, na forma:

VAL[i] := F( VAL1[ i-1 ], VAL2[ i-2 ], ... , VALn[ i-k ])

onde i é o índice da instância atual, i-1 o índice da instância anterior, etc., e i-k o índice da k-ésima instância anterior, tal que k seja mínimo e F dependa de VAL[ i-k ]. Chama-se k de arrasto. O arrasto da repetição é o maior arrasto encontrável nesta repetição. A seguir apresentamos alguns exemplos de valores de arrasto:

i- ao incializar um vetor atribuindo 0 a todos os elementos, o arrasto è 0, uma vez que esta inicialização independe de valores calculados em instâncias anteriores.

ii- ao calcular o somatório dos valores contidos em um vetor, o arrasto é 1, uma vez que a soma da instância atual depende da

soma parcial até a instância anterior

iii- ao calcular um novo elemento da série de Fibonacci, o arrasto é 2, uma vez que a fórmula de cálculo do i-ésimo elemento desta série é a[i]:=a[i-1]+a[i-2].

iv- no exemplo da figura 2 o arrasto é 2, uma vez que a variável de controle - número de registros lidos - é avançada de dois no caso de serem descartados registros em virtude da igualdade das chaves.

Intuitivamente pode-se dizer que o arrasto da repetição é igual ao número mínimo de ciclos necessários para que todas as variáveis e/ou estados modificados durante a repetição passem a depender exclusivamente de valores computados em ciclos anteriores. Ou seja, o arrasto da repetição é o número mínimo de ciclos necessário para esta repetição atingir o "steady state".

A determinação do arrasto de cada uma das repetições depende do entendimento do programa. Em geral o arrasto não é determinável por meios mecânicos simples. Sendo assim, é função do projetista determinar o arrasto de cada repetição contida no programa. Com o intuito de tornar mecânica a seleção do caminhos a serem percorridos durante os testes, sugere-se que os valores dos arrastos sejam introduzidos na expressão algébrica logo após ao "[" correspondente ao início da repetição. A figura 4 ilustra a expressão algébrica assim modificada.

A seguir formulamos o critério de seleção para repetições:

i- os casos {i = 0, 1, ..., arrasto} são casos especiais, e precisam ser testados um a um. São casos especiais, uma vez que correspondem a um estado transiente de inicialização do ciclo.

ii- em adição á inicialização é necessário, também, efetuar um teste para um número n genérico de iterações. É óbvio que desejamos o menor n possível com o intuito de minimizar o volume de dados e, consequentemente, minimizar o custo da execução do teste. Uma possível solução é utilizar um valor de n (número de ciclos a executar) tal que n > arrasto. Justifica-se este critério, observando que o arrasto é o menor número de iterações a ser realizado assegurando que todos os valores modificados durante a repetição tenham sido computados estritamente a partir de valores já computados em iterações anteriores desta repetição.

O critério acima procura simular a indução matemática. Ou seja, os arrasto primeiros testes formam a base da indução e o teste envolvendo

```
{ buffer vazio ao iniciar }

   gerar caminho( posição )

      caso token em posição

         fim de expressão algébrica:
            imprimir buffer

         rótulo de módulo:
            adicionar rótulo ao final do buffer
            gerar caminho( próxima posição )
            retirar rótulo do final do buffer

         "(":
            para cada  elemento i da alternativa  repetir
               computar posição inicial da i-ésima alternativa
               gerar caminho( posição calculada )
            fim repetir
            gerar caminho( posição após ")" )

         ")":
            /* não deveria ser encontrado */

         "[":
            para  i = 0, 1, ..., arrasto+1  repetir
               para  j de 1 até i  repetir
                  gerar caminho( posição após indicador de arrasto
               fim repetir
            fim repetir
            gerar caminho( posição após "]*" )

         "]*":
            /* faz nada, provoca retorno da recursão */

      fim caso

   fim gerar caminho
```

Figura 5. Algoritmo recursivo para a geração de caminhos visando cobertura completa.

n>arrasto ciclos simula o número "genérico" de repetições (hipótese de indução) ao qual é adicionado um novo ciclo "genérico" (passo de

indução). É óbvio que, sendo uma simulação utilizando um número definido não genérico, a validade no caso geral desta simulação não pode ser assegurada. Consequentemente, será possível identificar diversos exemplos de programas provadamente errados, para os quais o critério de seleção é incapaz de determinar um conjunto válido de casos teste (i.e. nenhum caso teste consegue determinar a existência do erro). Apesar desta potencial falibilidade do critério de seleção, não foi possível, até agora, encontrar um tal contra exemplo.

Existem diversas possíveis estratégias para selecionar os caminhos. Os casos extremos são:

i- cobertura completa, selecionam-se os caminhos para todas as possíveis combinações de alternativas e/ou número de ciclos a executar em cada uma das repetições.

ii- cobertura simples, selecionam-se suficientes caminhos de modo que cada alternativa e/ou repetição seja completamente testada, sem, no entanto, exigir o teste de combinações destas condições.

Deve estar claro que a cobertura completa gera o número máximo de caminhos cobrindo o programa, considerando o critério de seleção para repetições. Deve estar claro, ainda, que a cobertura simples gera o número mínimo de casos teste cobrindo todo o programa e satisfazendo o critério de seleção para repetições.

Neste artigo vamos nos restringir á cobertura completa. A figura 5 mostra o algoritmo utilizado para gerar todos os caminhos. A figura 6 ilustra o resultado da aplicação deste algoritmo ao problema da intercalação de arquivos sem repetição de chaves.

## 4. Avaliação do critério de seleção

Foram conduzidos diversos experimentos utilizando este critério de seleção de casos teste. Os resultados têm sido muito animadores, uma vez que o critério de seleção tem-se mostrado confiável, ou seja, efetivamente apontou erros sempre que havia falhas no programa sendo testado. No entanto, ainda são necessários muitos experimentos para poder-se determinar o grau de confiabilidade do presente critério de seleção.

O critério de seleção tem-se mostrado animador também sob outro ponto de vista, que é o de forçar o entendimento do programa por parte do examinador. Ou seja, a aplicação deste critério de seleção de casos teste induz o examinador a conduzir uma argumentação formal da correção do programa. Apesar de possivelmente incompleta, a argumentação tende a ser suficiente para assegurar ao examinador o entendimento preciso do

| | |
|---|---|
| A B | A e B vazios |
| A B C D G | A cnt 1, B vazio  cnt : contém |
| A B C E H | A cnt 1, B cnt 1, A=B |
| A B C F I | A vazio, B cnt 1 |
| A B C D G C D G | A cnt 2, B vazio |
| A B C D G C E H | A cnt 2, B cnt 1, B=A[2] |
| A B C D G C F I | A cnt 1, B cnt 1, A<B |
| A B C E H C D G | A cnt 2, B cnt 1, B=A[1] |
| A B C E H C D H | A cnt 2, B cnt 2, A[1]=B[1], A[2]=B[2] |
| A B C E H C F I | A cnt 1, B cnt 2, A=B[1] |
| A B C F I C D G | A cnt 1, B cnt 1, A>B |
| A B C F I C E H | A cnt 1, B cnt 2, A=B[2] |
| A B C F I C F I | A vazio, B cnt 2 |
| A B C D G C D G C D G | A cnt 3, B vazio |
| A B C D G C D G C E H | . |
| A B C D G C D G C F I | . |
| A B C D G C E H C D G | . |
| . . . | |
| A B C F I C F I C F I | A vazio, B cnt 3 |

Ao todo 3**3 + 3**2 + 3**1 + 3**0 = 40 casos

Figura 6. Caminhos a serem testados no programa da figura 2.

programa. A combinação de argumentação com testes tem-se mostrado altamente eficiente e eficaz como método de avaliação da qualidade do programa. É eficiente uma vez que o custo da argumentação dirigida não é demasiado, tampouco o é o custo de teste. É eficaz uma vez que o resultado é um programa altamente confiável.

O critério de seleção tem outra virtude que é a de ser hierárquico. Ou seja, é possível gerar casos teste para organizações modulares onde cada módulo é relativamente complexo, mas onde a estrutura é relativamente simples, i.e. contém poucos componentes. Desta forma o método se presta, em princípio, para apoiar a geração de testes de integração de programas complexos.

O maior defeito do critério de seleção é a sua tendência á explosão, i.e. geração de um número muito grande de casos teste. Para organizações modulares um pouco mais complexas (número elevado de repetições e/ou seleções) isto pode tornar inviável a aplicação econômica do critério de seleção. É verdade, entretanto, que o número de dados por caso teste tende a ser pequeno e, consequentemente, o custo de

cada execução de teste tende a ser também pequeno. Porém o grande número de casos teste pode tornar proibitivo o custo total.

Presentemente estão sendo estudados diversos caminhos para conter a explosão de casos teste:

i- identificar meios de eliminar caminhos a serem percorridos utilizando para isto informação especifica relativa às condições de repetição e/ou de seleção. Estas condições podem ser associadas ao diagrama de organização modular tal como ilustrado na figura 2. A aplicação deste mètodo de redução leva a uma execução simbólica parcial (envolvendo as condições).

ii- verificar se è possivel identificar de modo mecânico a independência de porções dos programa. Isto também resultarà numa redução de caminhos a testar uma vez que è desnecessàrio testarem-se combinações de condições no caso de regiões independentes.

iii- desenvolver armaduras de teste [Staa83] capazes de aplicarem os testes e confrontar os resultados esperados com os obtidos. Tais armaduras permitirão a execução automàtica dos diversos testes sem a necessidade de intervenção por parte do examinador.

Finalmente, está sendo examinado como estender o presente critério de seleção de modo a suportar recursão. Um caminho imaginado, mas ainda não examinado, è o de aplicar á recursão a mesma idèia que a utilizada para o caso de repetições. Isto levarà a diversas expressões regulares, uma para cada condição de percurso da recursão. A razão para se tentar este caminho è o fato da recursão levar a linguagens livres de contexto, ao invès de regulares, complicando assim, no gaso geral, a geração de expressões algèbricas capazes de sintetizar todos os caminhos possiveis no programa.

Referências bibliográficas

[Beizer83] Beizer, B.
Software Testing Techniques; New York; Van Nostrand Reinhold; 1983; 290 pags

[Gerhard77] Gerhard, S.L.; Yelowitz, L.
"Observations of the Fallibility in Applications of Modern Programming Methodologies"; in [Miller77]; 1977; pags 86-98

[Goodenough77] Goodenough, J.B.; Gerhard, S.L.
"Toward a Theory of Test Data Selection"; in [Miller77]; 1977; pags 68-85

[Howden80] Howden, W.E.
"Functional Program Testing"; in IEEE Transactions on Software Engineering SE-6(2); New York; março 1980; pags 162-169

[Jackson75] Jackson, M.A.
Principles of Program Design; New York; Academic Press; 1975

[Miller77] Miller, E. ed.
Tutorial on Program Testing Technigues; New York; IEEE ref. EHO 130-5; 1977

[Myers79] Myers, G.J.
The Art of Software Testing; New York; John Wiley; 1979; 177 pags

[Staa83] Staa, A.v.
Engenharia de Programas; Rio de Janeiro; Livros Técnicos e Científicos; 1983; 286 pags

[Yourdon79] Yourdon, E.; Constantine, L.L.
Structured Design: Fundamentals of a Discipline of Computer Program and System Design; Englewood Cliffs; Prentice-Hall; 1979