Testes de Caixa Branca e Caixa Preta

Prof. Pasteur Ottoni de Miranda Junior – PUC Minas
Disponível em www.pasteurjr.blogspot.com

1-Introdução

É hábito costumeiro do programador terminar um programa e em seguida iniciar uma bateria de testes com o objetivo de verificar correção e, muitas vezes, conformidade com determinados requisitos.Esta bateria de testes é, na maioria das vezes, realizada sem qualquer sistemática, sem análise criteriosa dos caminhos e/ou situações que deverão ser testadas.
A metodologia que aqui será apresentada tem por objetivo principal orientar na obtenção de **casos de teste** a serem utilizados para testar um determinado programa ou módulo de programa.. Estas técnicas não visam, portanto, orientar o programador em **como** testar. Muito menos determinam procedimentos para testar.
O objetivo primordial de qualquer técnica para testes é :

MAXIMIZAR (% defeitos encontrados/número de testes feitos)

As técnicas que vamos estudar auxiliam principalmente na minimização do número testes.
Deve-se deixar claro que testar não é a única maneira de se detectarem erros. As técnicas do tipo "walkthroughs" (revisões estruturadas) quando bem aplicadas chegam a detectar até 60% dos defeitos existentes.

2-Princípios sobre testes

Testar completamente é impossível. Por mais técnicas que empreguemos, dependendo da complexidade do programa sob teste, é impossível detectarem-se todos os erros nele existentes. É óbvio que isto não se aplica a programas do tipo "extremamente simples".

Testar é trabalho criativo e difícil. Ao contrário do que normalmente se pensa, testar não é fácil. Para se testar bem, é necessário que a pessoa responsável pelos testes tenha experiência e seja suficientemente treinada nas técnicas. Testar não é simples pelo seguinte:

-Para testar efetivamente, você tem que conhecer profundamente o sistema sob teste.
-Normalmente os sistemas não são simples e nem simples de entender.

Testes devem ser planejados e projetados. Todo processo de testar deve ser precedido de um planejamento dos casos de teste a serem utilizados. O que torna um teste realmente bom é a confiança que ele fornece, consequência do aumento da probabilidade de se descobrirem erros. Esta confiança, porém, tem um custo que, muitas vezes, ocasionado pela complexidade do programa, acaba se tornando alto. Assim sendo, um bom teste deve ser tal que seu custo seja minimizado, principalmente pela minimização do número de casos de teste necessários para se testar o programa.

Testar requer independência. Um testador independente é aquele dito não polarizado, ou seja, sem vícios ocasionados pelo conhecimento profundo do programa. Um testador polarizado tende a ignorar aspectos que tem certeza de estarem corretos, mas que muitas vezes podem conter erros. O testador independente tem como maior meta medir precisamente a qualidade do software.

3-Conceitos

Caso de Teste: Consiste em:

-Um conjunto de entradas possíveis do programa.

-Um conjunto de saídas esperadas para as entradas.

Por exemplo:
Seja um programa para classificar triângulos com a seguinte especificação:

ENTRADA: Os três lados tal que a >= b >= c (a,b,c inteiros)
SAÍDA: Classificação do triângulo:

1- Não é triângulo
2-Triângulo equilátero
3- Triângulo isósceles
4- Triângulo escaleno reto
5- Triângulo escaleno obtuso
6- Triângulo escaleno agudo

Temos os seguintes casos de teste:

SAÍDA	ENTRADA
Tripla ilegal	(1,2,3) (a < b < c)
Não é triângulo	(11,6,4)
Triângulo equil..	(1,1,1)
Triângulo isósc.	(2,2,1)
Triângulo esc. reto	(5,4,3)
Triângulo esc. agudo	(6,5,4)
Triângulo esc. obtuso	(4,3,2)

Estes pares entrada-saída (casos de teste) poderiam ser utilizados para testar o nosso programa. Porém, tais casos de teste foram gerados sem qualquer sistemática.

Teste: É um conjunto de casos de teste a serem aplicados em um programa.

Procedimentos de Teste: É a sequência de ações para se executar um teste ou um caso de teste.

Métodos de teste: São métodos para se **gerarem** casos de teste.
Classificação:

-Métodos de caixa branca- Nesta metodologia os casos de teste são gerados tendo-se conhecimento da estrutura interna (lógica) do programa. São também denominados **testes estruturais.** São os seguintes os que iremos estudar:

-Cobertura Lógica (Critérios Myers):

-Método dos Caminhos Básicos.

-**Métodos de caixa preta-** Nesta metodologia os casos de teste são gerados sem o conhecimento da estrutura interna do programa. Apenas o conhecimento das entradas e saídas possíveis para o programa é necessário. São também denominados **testes funcionais**. Estudaremos os seguintes métodos:

-Partição em Classes de Equivalência

-Grafos de Causa-Efeito

-Teste de Condições de Fronteira.

4-Métodos de Caixa Branca

4.1 - Cobertura Lógica (Critérios Myers)

Este método é constituído de critérios que, quando obedecidos, determinam os casos de teste a serem gerados. Os critérios vão se tornando cada vez mais abrangentes e rigorosos, partindo-se do mais simples, ineficiente e menos rigoroso (Cobertura de Comandos) até o mais complexo, eficiente e rigoroso (Cobertura de Múltiplas Condições).
A escolha do critério adequado será norteada pelos seguintes fatores:

-Complexidade do programa a testar: Programas mais simples podem ser satisfeitos pela utilização de critérios menos rigorosos.Muitas vezes a utilização de critérios mais rigorosos em programas muito simples acaba por onerar excessivamente o processo de testes.

-Estrutura do programa a ser testado. Certas estruturas são mais adequadas a determinados critérios. Por exemplo, programas ricos em comandos, podem ser testados utilizando-se o critério de cobertura de comandos, programas com decisões complexas devem ser testados por cobertura de decisões /condições.

-Criticidade do programa a testar: Programas cujo funcionamento não é crítico podem exigir testes menos rigorosos. Portanto, critérios menos rigorosos podem ser suficientes para se testar o programa.

-Nível de confiança que se deseja atingir. Se o nível de confiança de bom funcionamento é alto,critérios mais rigorosos são necessários.

4.1.1 - Cobertura de Comandos

Neste critério os casos de teste devem ser gerados de forma que ao serem executados, o fluxo do programa passe por todos os COMANDOS existentes no mesmo.

Por exemplo:
Seja o fluxograma da fig. 1:

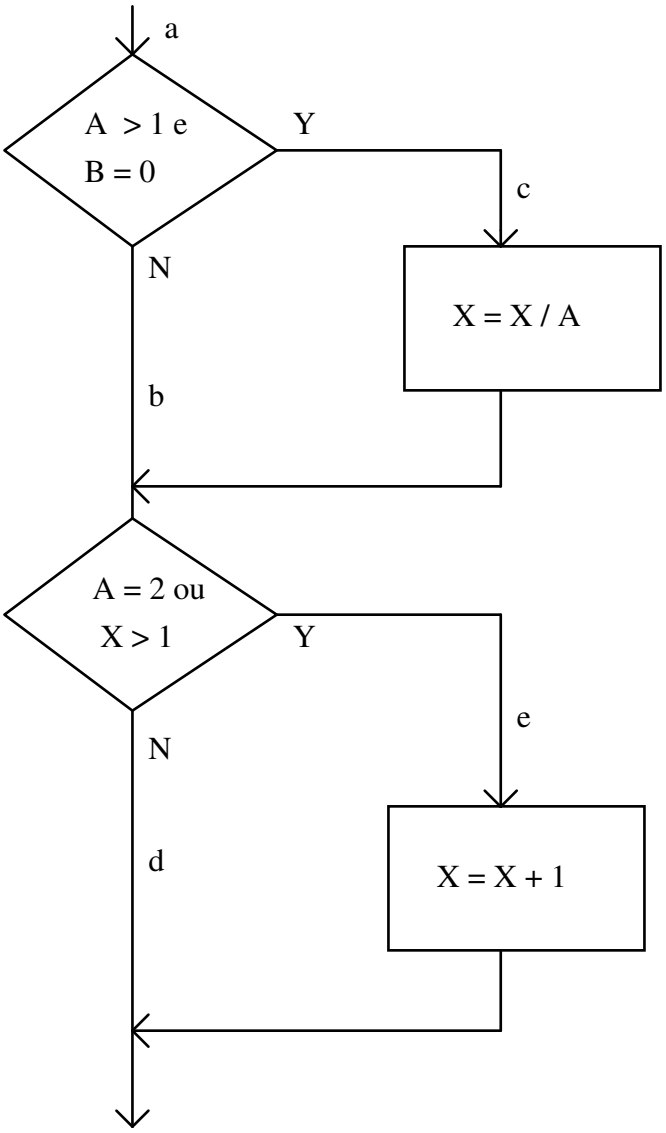


Fig. 1 - Fluxograma a utilizar nos exemplos

Um único caso de teste é suficiente para que este critério seja satisfeito :

A = 2, B=0 , X qualquer (caminho a c e, passa por todos os comandos, quais sejam: X = X/ A e
X= X + 1)

O critério é satisfeito, porém, podemos verificar que mesmo para um programa simples como este ele falha em algumas situações. Se por exemplo, houvesse os seguintes erros no programa:

A > 1 OU (no lugar de E) B= 0
A=2 OU X > 0 (no lugar de 1)

O caso de teste não detectaria, pois o fluxo do programa continuaria passando por a c e. A saída do programa estaria inalterada, não denunciando, portanto, o erro cometido.

4.1.2-Cobertura de Decisões

Neste critério os casos de teste serão gerados de tal forma que cada decisão tenha todas as suas opções de saída (V ou F) percorridas ao menos uma vez. Este critério engloba o anterior.
No exemplo da fig. 1, dois casos de teste são suficientes para satisfazer este critério:

A=3, B=0, X=3 faz o fluxo do programa passar por a c d
A=2, B=1, X=1 faz o fluxo do programa passar por a b e

Os erros exemplificados no critério anterior seriam detectados.
Porém, observe que se na segunda decisão houvesse um erro como X < 1 ao invés de X > 1, a saída do programa permaneceria inalterada, não permitindo a detecção do erro.

4.1.3-Cobertura de Condições

Neste critério os casos de teste são tais que cada condição em uma decisão deve ser testada ao menos uma vez em todas as suas saídas possíveis.

No exemplo da fig.1 temos as seguintes condições:

A> 1, B=0, A=2, X > 1

Temos que gerar casos de teste para testar as situações :

A>1 e A <=1
B=0 e B <> 0
A=2 e A <> 2
X>1 e X <= 1

Os seguintes casos de teste satisfazem a estas situações:

A= 2, B=0, X=3
A=2, B= 1, X=1

Observe que com este critério conseguiríamos detectar o erro descrito no critério anterior.
Porém ocorre que nem todas as saídas das decisões são percorridas.

4.1.4 - Cobertura de Decisões-Condições

É uma combinação dos dois últimos critérios. Cada condição em uma decisão é testada em todas as suas saídas possíveis e cada decisão tem sua saída V ou F percorrida ao menos uma vez.

Ainda para o fluxograma da fig. 1 temos que os seguintes casos de teste satisfazem este critério:

A= 1, B=1, X= 1 (satisfaz A <= 1, B<> 0 e X <=1, fazendo com que o fluxo do programa passe pelo N das duas decisões)

A= 2, B=0, X= 3 (satisfaz A > 1, B = 0 e X > 1, fazendo com que o fluxo do programa passe pelo Y das duas decisões)

Ocorre que, em algumas situações, uma condição pode mascarar outras.

4.1.5-Cobertura de múltiplas condições

Os casos de teste são tais que todas as combinações de condições em cada decisão são cobertas.

Combinações de condições a serem cobertas:

1- A > 1, B= 0	5- A=2, X> 1
2- A > 1, B<>0	6- A=2, X<=1
3- A <= B= 0	7- A<>2, X> 1
4- A > =1 B= 0	8- A<>2, X<= 1

Os casos de teste que cobrem estas combinações são:

A=2,B=0,X=5 cobre 1,5
A=2,B=1,X=1 cobre 2,6
A=1,B=0,X=2 cobre 3,7
A=1,B=1,X=1 cobre 4,8

Este é sem dúvida o critério mais abrangente. Porém esta abrangência tem um preço: o número de casos de teste para satisfazê-lo é maior.
As falhas deste critério estão relacionadas ao número de caminhos percorridos: ele não garante que este número seja suficiente para testar com confiança o programa.

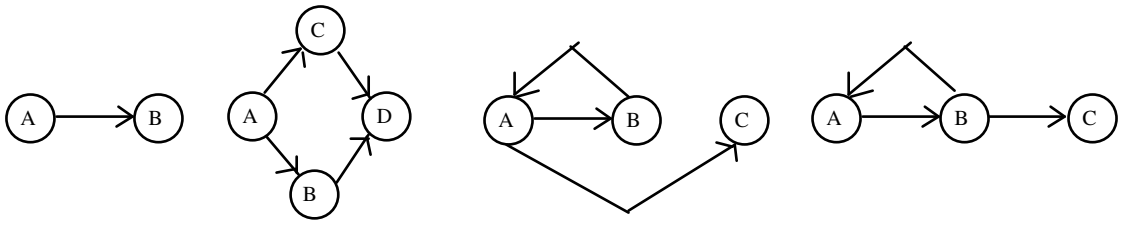
4.2 - Método dos Caminhos Básicos

4.2.1 - Introdução

Método criado pelo matemático norte-americano Thomas McCabe baseado na teoria de grafos.
Sua lógica consiste essencialmente em fazer com que os casos de teste sejam gerados de forma a fazer com que o fluxo do programa passe por um número mínimo de caminhos entre a entrada e a saída do programa, sem o risco de ocorrerem redundâncias.

4.2.2- Grafo de Controle

É um grafo orientado (ou seja, suas arestas possuem setas) que descreve o fluxo de controle do programa. Na realidade, o grafo de controle não passa de um fluxograma com símbolos diferentes, quais sejam:



Sequência

if A
then B
else C;

While A
do B;

repeat
do A
until B

A estrutura CASE é idêntica à de seleção com mais opções

Fig. 2 - Estruturas do grafo de controle

Por exemplo, o procedimento:

```
procedure Teste(A,B,C : integer);

begin
if A=B then
        A : = A + 1
else
        if B = C then
                B : = B+1
        else
                B : = B -1;

end;
```

teria o seguinte grafo de controle:

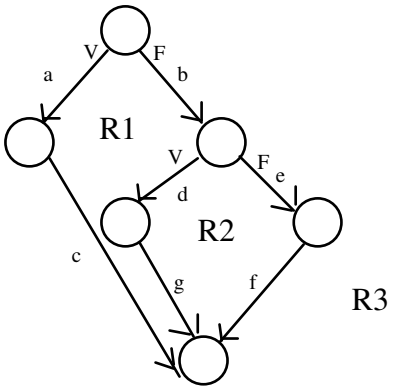


Fig. 3- Grafo de controle

4.2.3-Caminhos independentes

No grafo de controle identificamos os chamados caminhos independentes. Dentre um conjunto de caminhos identificados para um grafo, dizemos que eles são independentes quando nenhum deles é formado da combinação de dois ou mais outros.

Para determinarmos os caminhos independentes de um grafo, temos que inicialmente determinar quantos são. Existem 3 formas de se determinar este número, também denominado **complexidade ciclomática**:

-Contar o número de regiões no grafo de controle. No exemplo da figura 3, temos três regiões, R1, R2 e R3: as duas internas(R1 e R2) e a externa (R3). Logo, este grafo tem 3 caminhos independentes.
-Aplicar a fórmula:

$$C = E - N + 2$$

onde C = número de caminhos independentes
E = número de arestas
N = número de nós

No exemplo da figura 3 temos E = 7, N = 6 , logo C = 7-6+2 = 3

-Contar o número de estruturas de decisão no programa e somar 1. No exemplo da figura 3, temos 2 "if". Logo, o número de caminhos independentes é 3.

Para identificar os caminhos independentes em um grafo fazemos o seguinte:

-Determinar o número de caminhos independentes conforme visto anteriormente.
-Tomar o caminho mais a esquerda possível no grafo como primeiro caminho lindependente. No exemplo da fig. 3, este caminho é o **ac.**
-Tomar o próximo caminho à direita, tendo o cuidado de incluir nele pelo menos uma aresta que não tenha sido incluída nos outros caminhos já determinados. No exemplo, caminho **bdg.**
-Repetir o passo anterior até que se obtenham todos os caminhos. No exemplo, só nos resta o caminho **bef.**

4.2.4 - Geração dos casos de teste

Cada caminho obtido dará origem a um caso de teste. Os casos de teste são gerados de forma que façam com que os caminhos aos quais correspondam sejam percorridos. No exemplo da figura 3 temos:

Caminho ac: Entradas: A=3, B=3, C=3. Saídas: A=4, B=3.
Caminho bdg: Entradas: A=3, B=4, C=4. Saídas: A=3, B=4.
Caminho bef: Entradas: A=3, B=5, C=6. Saídas: A=3, B=4.

4.2.5-Passos do método

Passo 1: Desenhar o grafo de controle
Passo 2: Determinar o número de caminhos independentes (complexidade ciclomática)
Passo 3: Obter os caminhos independentes
Passo 4: Para cada caminho, gerar os casos de teste, utilizando a especificação do programa para inferir os resultados esperados.
Passo 5:
-Executar cada caso de teste
-Checar os resultados obtidos contra os esperados
-O teste estará completo quando houver confiança absoluta em que todos os casos de teste obtêm resultados satisfatórios.

5-Testes de Caixa Preta

5.1 - Partição em Classes de Equivalência

5.1.1-Introdução

Esta metodologia é adequada ao teste de valores típicos de entrada de um programa. Os casos de teste são gerados a partir do conhecimento das entradas , de maneira sistemática e direta.

5.1.2 Definição - Classe de Equivalência

Uma classe de equivalência nada mais é do que um subconjunto das entradas possíveis de um programa, de tal forma que um teste efetuado para um valor representativo da mesma é equivalente ao teste para qualquer outro valor valor nesta classe.

5.1.3-Passos do método

1)Definir as classes de equivalência

1.1)Definir as chamadas condições de entrada
Definição: Condição de entrada. Uma condição de entrada é qualquer intervalo de entradas válidas de um programa. Tipos:

a)Faixa de valores. Ex.: 1 < ITEM <1000

b)A entrada envolve um certo número de valores. Ex: Um vetor de até 6 elementos

c)Conjunto de Valores. Ex: Conjunto dos veículos motorizados = {caminhão,carro,ônibus}

d)Condição booleana. Ex: O primeiro caracter de um identificador deve ser uma letra

1.2)Definir as classes de equivalência propriamente ditas, que podem ser válidas ou inválidas. Estas classes são definidas a partir da condição de entrada.Ex:

a) 1< ITEM < 1000 Classe válida : 1 <ITEM < 1000
Classes inválidas : ITEM <= 1 e ITEM >= 1000
b)Vetor de até 6 elementos. Classe válida: vetor com 4 elementos
Classes inválidas: Vetor com zero elemento e Vetor com 7 elementos
c)Conjunto de valores. Classe válida: {caminhão, carro, ônibus}

Classe inválida : {carroça}

d)Classe válida: O primeiro caracter do identificador é uma letra

Classe inválida: O primeiro caracter do identificador não é uma letra

1.3)Se for constatado que todos os elementos de uma classe não podem ser tratados da mesma forma, dividir a mesma em outras classes.

2)Identificar casos de teste

2.1)Numerar as classes válidas e inválidas

2.2)

-Cada classe inválida gera um caso de teste (ou seja, gerar a entrada tal que cubra a classe inválida).

-Para as classes válidas, cada caso de teste é gerado de forma a cobrir o maior número possível delas

5.1.4-Exemplo

Suponhamos que se queira testar um compilador (verifcador de sintaxe) para o seguinte comando:

INSERT VALUE / MNEMONIC = <nome do mnemônico> / VALUE = <valor>

Restrições:

-Comando só aceita maiúsculas

-<nome do mnemônico> provém de uma tabela de 1168 mnemônicos

-Tamanho do mnemônico é no máximo 8

-<valor> = faixa de validade do mnemônico

Condições de entrada	Tipo de condição	Classes válidas	Classes inválidas
Comando só tem maiúscula	Booleana	sim (1)	Não (2)
Tamanho do menmônico	Faixa	1-8 (3)	0, (4) > 8 (5)
Mnemônico válido	Booleana	sim (6)	não (7)
Valor do mnem. ABSBRG ..	Faixa	0-360 (8)	< 0 (9) >360 (10)
Valor do mne. XTKTE004	Faixa	-30..+30 (11)	< -30 (12) > +30 (13)
Próximo token após insert é VALUE ...	Booleana	Sim (14)	Não (15)

Casos de teste:

Para as classes válidas, pegar o maior número delas com um número mínimo de casos de teste:

INSERT VALUE / MNEMONIC = ABSBRG / VALUE = 30 => Pega as classes número 1,3,6,8,14

INSERT VALUE / MNEMONIC = XTKTE004 / VALUE = -2.5 => Pega as classes número 1,3,6,11,14

Para as classes inválidas: um caso de teste para cada.

Classe 2: INSERT VALUE / mnem = ABSBRG / VALUE = 30
Classe 4: INSERT VALUE / MNEM = / VALUE = 30
Classe 5: INSERT VALUE / MNEM = XUSASAASASAS / VALUE = 30
Classe 7: INSERT VALUE / MNEM = AAX / VALUE = 30 (Supondo que AAX não esteja na tabela)
Classe 9: INSERT VALUE /MNEM = ABSBRG / VALUE = -35
Classe 10: INSERT VALUE / MNEM = ABSBRG / VALUE = 400
Classe 12: INSERT VALUE / MNEM = XTKTE004 / VALUE = -50
Classe 13: INSERT VALUE / MNEM = XTKTE004 / VALUE = 60
Classe 15: INSERT VALOR / MNEM = XUSASAASASAS / VALUE = 30

Obviamente o exemplo acima não inclui todos os 1168 mnemônicos existentes . Na prática teríamos que gerar os casos de teste para todos eles. O exemplo também não inclui outras verificações sintáticas, como por exemplo, depois da palavra VALUE vir a barra (/).

5.2 Método do Grafo de Causa e Efeito

5.2.1-Introdução

Método baseado em especificações de **ENTRADAS (causas) e SAÍDAS (efeitos)**.
Representa a combinação de entradas em saídas de forma bastante representativa, não redundante e, muitas vezes, de tamanho minimizado. Além disso possui uma vantagem extra, pois aponta ambiguidades e incompletezas na especificação. Porém, quando o grafo de causa-efeito torna-se muito complexo, fica bastante difícil a geração da tabela de decisão. Nesta situação, a utilização deste método só é viável com o auxílio do computador.

5.2.2 -O Grafo de Causa-Efeito

É uma rede lógica combinatorial que representa a especificação do programa. É análogo a um circuito lógico digital.
Para construí-lo procedemos da seguinte forma

-Identificar as causas : as causas são as entradas do programa. Tomar o cuidado de não identificar causas complementares, como por exemplo "x > 60" e "x <=60". Colocá-las à esquerda no diagrama.

-Identificar efeitos: os efeitos são as saídas do programa. Colocá-los à direita no diagrama.

-Relacionar as causas com os efeitos através da seguinte notação:

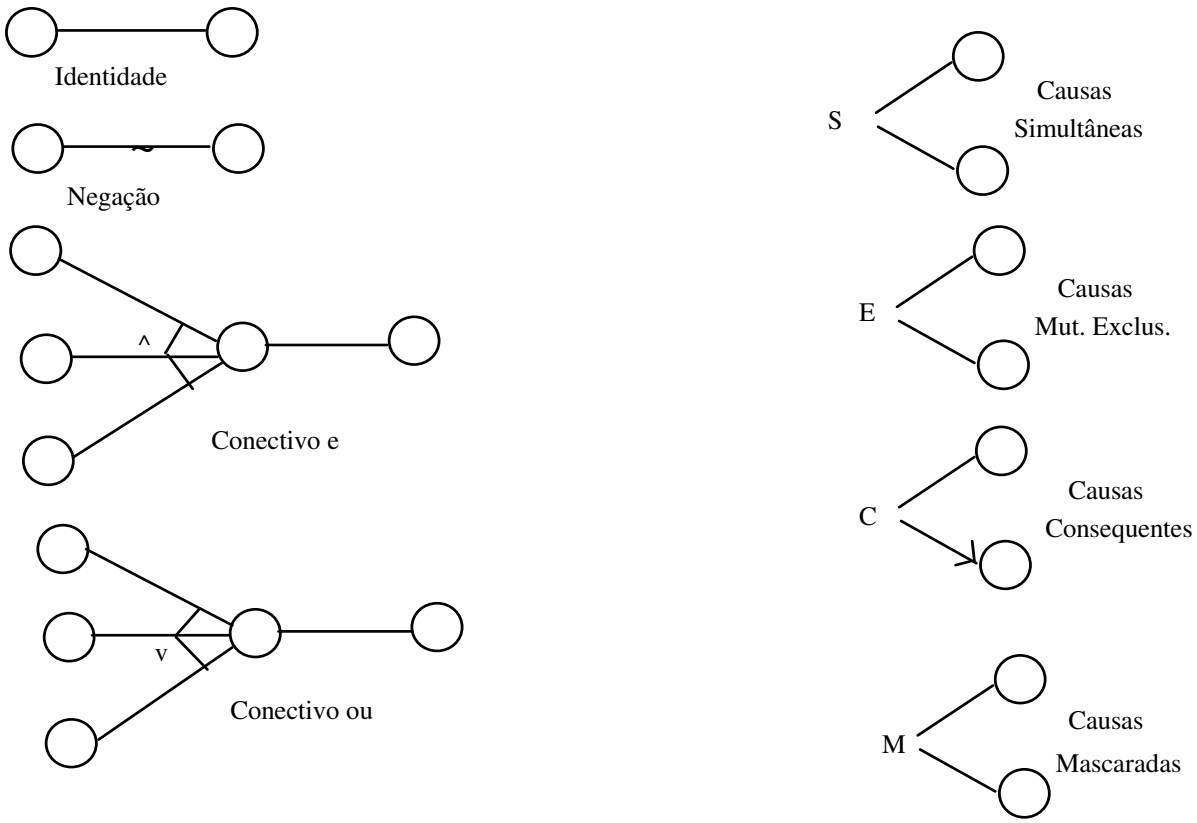


Fig. 4 Notação para conexões intercausas

5.2.3-Passos do Método

1)Desenhar o grafo
2)Transformar o grafo em tabela de decisão. Como? Para cada um dos efeitos, gerar combinações entre causas que façam com que sejam ativados.
3)Cada combinação de causas geradas no passo 2) vai dar origem a um caso de teste

5.2.4 - Exemplo

Seja a seguinte especificação para um compilador de um comando bem simples:

O caracter na coluna 1 deve ser "A" ou "B" . O caracter na coluna 2 deve ser um número. Se isto ocorrer, o comando está correto. Se o primeiro caracter está incorreto, emitir mensagem X12. Se o segundo caracter não é um dígito, emitir mensagem X13.

Causas identificadas:

São as entradas possíveis ao programa:

1-Caracter "A" na primeira coluna

2-Caracter "B" na primeira coluna
3-Caracter na coluna 2 é dígito
Observe que não colocamos, por exemplo, a causa "caracter na coluna 2 não é dígito", pois esta causa é complementar à causa 3, sendo obtida por negação desta.

Efeitos identificados:

70- Comando correto
71- Mensagem X12
72-Mensagem X13

O grafo obtido é o seguinte:

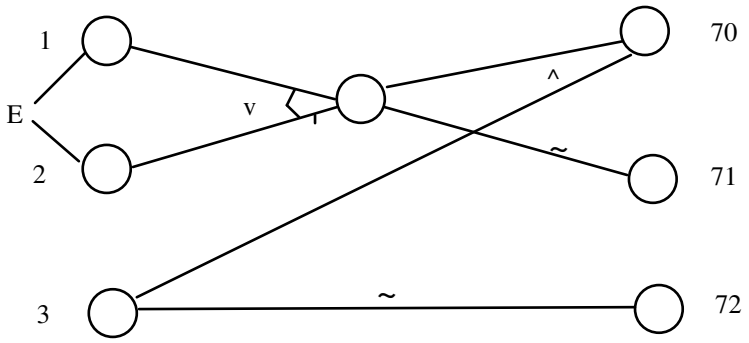


Fig. 5 - Grafo de Causa-Efeito do exemplo

A tabela de decisão é a seguinte.

1	1	0	0	X
2	0	1	0	X
3	1	1	X	0
70	1	1	0	0
71	0	0	1	0
72	0	0	0	1

Repare que, ao invés de gerarmos 8 combinações entre as entradas (todas as possíveis), obtivemos apenas 4, que são coerentes com a especificação do programa. Por este método, não precisamos testar todas as combinações.

Cada coluna de combinação da tabela vai então gerar um caso de teste:

PRIMEIRO CARACTER	SEGUNDO CARACTER	SAÍDA
A	1	Comando Correto

B	1	Comando Correto
X	3	Mensagem X12
X	B	Mensagem X13

5.3 - Análise de Valores de Fronteira

5.1 - Introdução

Quando testamos um determinado programa, é muito comum utilizarmos apenas valores típicos da entrada ou saída do mesmo. Porém, a experiência mostra que testes que exploram **condições de fronteira** dão mais retorno que testes que não o fazem. O chamado "Princípio da Timidez" postula o seguinte:

"Os 'BUGS' se concentram nos cantos e nas frestas"

Este método complementa o Partição em Classes de Equivalência (que utiliza valores típicos), testando valores focalizados nas fronteiras dos intervalos de validade. Na realidade não existe uma metodologia para se testar nas fronteiras, e sim diretrizes para tal.

5.2 - Diretrizes

1)Para cada **faixa de valores** (na entrada ou na saída), se os extremos da faixa são os valores **a** e **b ,** geramos os seguintes casos de teste:

-Entrada sobre a (sobre a fronteira inferior)
-Entrada sobre a - um valor bem pequeno (um pouco abaixo da fronteira inferior)
-Entrada sobre b (sobre a fronteira superior)
-Entrada sobre b + um valor bem pequeno (um pouco acima da fronteira superior)

2)Para estruturas do tipo **conjunto ordenado** (arquivo sequencial, lista tabela), gerar os casos:

-Conjunto vazio
-Conjunto com 1 elemento
-Conjunto cheio
-Conjunto cheio + 1

3)Quando possível, gerar casos de teste que levem cada saída para zero (ou nulo)

4)Buscar outras situações limites específicas do problema, por exemplo:

Suponha um programa cuja saída é uma lista de itens. Alguns casos de fronteira seriam:

O número de itens na lista é tal que:
A)Encha uma página (sai alguma página extra?)
B)Encha uma página + 1 item
C)Um item apenas
D) Zero item (Sai cabeçalho?)

ESTUDO DIRIGIDO

1)Seja o seguinte procedimento:

```
Procedure ResultFin(   NOTA, FREQUENCIA : real);

begin
readln(NOTA, FREQUENCIA);
if  (NOTA >= 60) then  begin
        if (FREQUENCIA >= 75) then begin
                writeln('APROVADO');
                readln;
        end
        else if (FREQUENCIA > =50) then begin
                writeln('RECUPERACAO');
                readln;
              end
                 else  begin
                writeln('REPROVADO');
                readln;
               end;
end
else if  (NOTA >=40) then begin
        writeln('RECUPERACAO');
        readln;
      end
       else begin
        writeln('REPROVADO');
        readln;
       end;
end.
```

a)Fazer um grafo de causa-efeito para este programa.
b)Gerar a tabela de decisão e os casos de teste

2)Para o procedimento do exercício anterior faça:

a)Desenhe o grafo de controle
b)Determine a complexidade ciclomática eos caminhos independentes
c)Determine os casos de teste

3)Seja a seguinte especificação de sintaxe:

COPY -------->> ORIGEM ------->> DESTINO ----->> > ------>> DIRECIONAMENTO

ORIGEM E DESTINO : são arquivos no formato NOME.EXTENSÃO, onde NOME tem um máximo de 8 caracteres alfanuméricos, exceto "[" , "]", "/", ",", ". ". EXTENSÃO tem um máximo de 3 caracteres e as mesmas exceções de NOME.. DIRECIONAMETO,PODE SER :

PRN (impressora)
NULL (sem ecoar)
ARQUIVO (com as mesmas definições deORIGEM e DESTINO)

a)Gerar as classes de equivalência válidas e inválidas.
b) Gerar os respectivos casos de teste.

REFERÊNCIAS

MYERS, G.F. *The Art of Software Testing.* 1a edição, John Wiley and Sons, 1979.