4 Desenho de casos de teste

Para além das questões psicológicas discutidas no Capítulo 2, a consideração mais importante nos testes do programa é a concepção e criação de casos de teste eficazes.

Os testes, por mais criativos e aparentemente completos, não podem garantir a ausência de todos os erros. A concepção do caso de teste é tão importante porque é impossível fazer testes completos. Dito de outra forma, um teste de qualquer programa deve ser necessário - essencialmente incompleto. A estratégia óbvia, portanto, é tentar tornar os testes o mais completos possível.

Dadas as limitações de tempo e custo, a questão chave dos testes torna-se:

> Qual de todos os casos de teste possíveis tem a maior probabilidade de detectar o maior número de erros?

O estudo de metodologias de concepção de casos de teste fornece respostas a esta questão.

Em geral, a metodologia menos eficaz de todas é o teste aleatório de

entrada - o processo de testar um programa seleccionando, aleatoriamente, algum subconjunto de todos os valores de entrada possíveis. Em termos da probabilidade de detectar a maioria dos erros, uma colecção de casos de teste seleccionados aleatoriamente tem poucas hipóteses de ser um subconjunto óptimo, ou mesmo próximo do óptimo. Portanto, neste capítulo, queremos desenvolver um conjunto de processos de pensamento que permitam seleccionar os dados de teste de forma mais inteligente.

Myers, G. J., Sandler, C., & Badgett, T. (2011). A arte de testar software. J o h 4 n W1iley & Sons, Incorporated.

O capítulo 2 mostrou que os testes exaustivos da caixa negra e da caixa branca são, em geral, impossíveis; ao mesmo tempo, sugeriu que uma estratégia de teste razoável poderia conter elementos de ambos. Esta é a estratégia desenvolvida neste capítulo. É possível desenvolver um teste razoavelmente rigoroso utilizando certas metodologias de concepção de casos de teste orientados para a caixa negra e, em seguida, suplementos nestes casos de teste, examinando a lógica do programa, utilizando os métodos da caixa branca.

As metodologias discutidas neste capítulo são as seguintes:

Caixa Negra	Caixa Branca
Divisória de equivalência	Cobertura de equivalência
Análise do valor-limite	Cobertura da decisão
Gráficos de causa-efeito	Cobertura da condição
	Adivinhação de erro
	Decisão/condição cobertura
	Cobertura de condições múltiplas

Embora discutiremos estes métodos separadamente, recomendamos que utilize uma combinação da maioria, se não de todos, para conceber um teste rigoroso de um programa, uma vez que cada método tem pontos fortes e fracos distintos. Um método pode encontrar erros, outro método pode ignorar, por exemplo.

Nunca ninguém prometeu que os testes de software seriam fáceis. Para citar um velho sábio, "Se pensou que conceber e codificar aquele programa era difícil, ainda não viu nada".

O procedimento recomendado é desenvolver casos de teste utilizando os métodos da caixa negra e depois desenvolver casos de teste suplementares, conforme necessário, com métodos da caixa branca. Discutiremos primeiro os métodos de caixa branca mais amplamente conhecidos.

Teste da White-Box

O teste da caixa branca diz respeito ao grau em que os casos de teste

exercem - cise ou cobrem a lógica (código fonte) do programa. Como vimos no Cap- ter 2, o derradeiro teste da caixa branca é a execução de cada caminho no programa; mas o teste completo do caminho não é um objectivo realista para um programa com loops.

Teste de Cobertura Lógica

Se se afastar completamente dos testes do caminho, pode parecer que um objectivo digno seria executar todas as declarações do programa pelo menos uma vez. Infelizmente, este é um critério fraco para um teste razoável da caixa branca. Este conceito é ilustrado na Figura 4.1. Assumir que esta figura representa um pequeno programa a ser testado. Segue-se o snippet do código Java equivalente:

```
public void foo(int A,int B,int X) {
    if(A> 1 && B¼¼0) {
        X¼X/A;
    }
    if(A¼¼2 jj X> 1) {
        X¼Xþ1;
    }
}
```

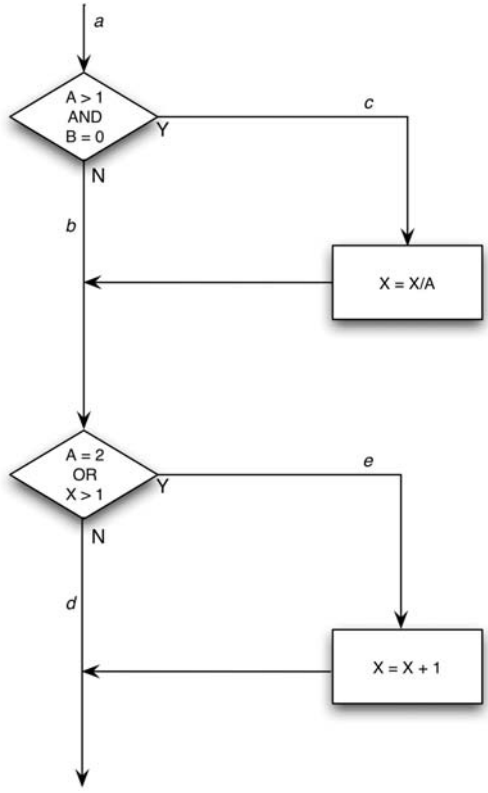


FIGURA 4.1 Um pequeno programa a ser testado.

Poderia executar cada declaração escrevendo um único caso de teste que atravessa o *ás do* caminho. Ou seja, definindo A¼2, B¼0, e X¼3 no ponto a, cada declaração seria executada uma vez (na verdade, a *X* poderia ser atribuído qualquer valor inte- ger > 1).

Infelizmente, este critério é um critério bastante pobre. Por exemplo, talvez a primeira decisão deva ser uma ou mais do que uma e. Se assim fosse, este erro não seria detectado. Talvez a segunda decisão devesse ter indicado X> 0; este erro não seria detectado. Além disso, existe um caminho através do programa no qual *X* passa inalterado (o caminho *abd*). Se isto fosse um erro, não seria detectado. Por outras palavras, o critério de cobertura da declaração é tão fraco que geralmente é inútil.

Um critério de cobertura lógica mais forte é conhecido como cobertura de decisão ou cobertura de filial. Este critério estabelece que se deve escrever casos de teste suficientes para que cada decisão tenha um resultado verdadeiro e falso, pelo menos uma vez. Por outras palavras, a direcção de cada ramo deve ser percorrida pelo menos uma vez. Exemplos de declarações de ramo ou de decisão são as declarações de *mudança de caso*, de *"do-while"*, e de *"if-else"*. As declarações multi-caminho *GOTO* qualificam-se em algumas linguagens de programação, tais como Fortran.

A cobertura de decisões pode normalmente satisfazer a cobertura de declarações. Uma vez que cada declaração está em algum subcaminho que emana quer de uma declaração de ramo quer do ponto de entrada do programa, cada declaração deve ser executada se cada direcção de ramo for executada. Existem, no entanto, pelo menos três excepções:

- Programas sem decisões.
- Programas ou sub-rotinas/métodos com múltiplos pontos de entrada. Um dado
  só pode ser executado se o programa for introduzido num determinado ponto de entrada.
- Declarações dentro das *ON-unidades*. Atravessando todas as direcções dos ramos
  não necessariamente causar a execução de todas as *ON-unidades*.

Uma vez que consideramos a cobertura de declarações como uma condição necessária, a cobertura de decisões, um critério aparentemente melhor, deve ser definido para incluir a cobertura de declarações. Assim, a cobertura de decisão requer que cada decisão tenha um resultado verdadeiro e falso, e que cada declaração seja ex - algemada pelo menos

uma vez. Uma forma alternativa e mais fácil de o expressar é que cada decisão tenha um resultado verdadeiro e falso, e que cada ponto de entrada (incluindo `as ON-unidades`) seja invocado pelo menos uma vez.

Esta discussão considera apenas decisões ou ramificações de duas vias e tem de ser modificada para programas que contenham decisões multi-caminhos. Exemplos são os programas Java que contêm declarações *em caso de mudança*, os programas Fortran com aritmética de matizes (três vias) *IF* ou declarações *GOTO* computorizadas ou aritméticas, e os programas COBOL que contêm declarações *GOTO* alteradas ou declarações *GO-TO-DEPENDING-ON*. Para tais programas, o critério é exercer - citando cada resultado possível de todas as decisões pelo menos uma vez e invocando cada ponto de entrada para o programa ou sub-rotina pelo menos uma vez.

Na Figura 4.1, a cobertura de decisão pode ser cumprida por dois casos de teste cobrindo os caminhos *ace* e *abd* ou, alternativamente, *acd* e *abe*. Se escolhermos a última alternativa, os dois casos de teste são A¼3, B¼0, X¼3 e A¼2, B¼1, e X¼1.

A cobertura de decisões é um critério mais forte do que a cobertura de declarações, mas ainda assim é bastante fraca. Por exemplo, há apenas 50% de hipóteses de explorarmos o caminho onde *x* não é alterado (ou seja, apenas se escolhermos a primeira alternativa). Se a segunda decisão estivesse em erro (se tivesse dito X< 1 em vez de X> 1), o erro não seria detectado pelos dois casos de teste do exemplo anterior.

Um critério que por vezes é mais forte do que a cobertura de decisões é a condi- ção de cobertura. Neste caso, escreve-se casos de teste suficientes para assegurar que cada condição numa decisão toma todos os resultados possíveis pelo menos uma vez. Mas, tal como na cobertura de decisão, isto nem sempre leva à execução de cada declaração, pelo que um acréscimo ao critério é que cada ponto de entrada no programa ou sub-rotina, bem como as *unidades ON*, sejam invocadas pelo menos uma vez. Por exemplo, a declaração de ramificação:

```
DO K¼0 a 50 WHILE (JþK< QUEST)
```

contém duas condições: K é inferior ou igual a 50, e JþK é inferior à QUEST? Assim, seriam necessários casos de teste para as situações K<¼50, K> 50 (para atingir a última iteração do laço), JþK< QUEST, e JþK>¼QUEST.

A figura 4.1 tem quatro condições: A> 1, B¼0, A¼2, e X> 1. Assim, são necessários casos de teste suficientes para forçar as situações em que A> 1, A<¼1, B¼0, e B<> 0 estão presentes no ponto a e onde A¼2, A<> 2, X> 1, e X<¼1 estão presentes no ponto b. Um número suficiente de casos de teste que satisfazem o critério, e os caminhos percorridos por cada um deles, estão presentes:

```
A¼2, B¼0, X¼4 ace
```

A¼1, B¼1, X¼1 *adb*

Note-se que embora o mesmo número de casos de teste tenha sido gerado para este exemplo, a cobertura da condição é normalmente superior à cobertura da decisão, na medida em que pode (mas nem sempre) fazer com que cada condição individual de uma decisão seja executada com ambos os resultados, ao passo que a cobertura da decisão não o faz. Por exemplo, na mesma declaração de ramificação

```
DO K¼0 a 50 WHILE (JþK< QUEST)
```

é um ramo de duas vias (executar o corpo do laço ou ignorá-lo). Se estiver a utilizar testes de decisão, o critério pode ser satisfeito deixando o laço correr de `K¼0 a 51`, sem nunca explorar a circunstância em que a cláusula `WHILE` se torna falsa. Com o critério de condição, contudo, seria necessário um caso de teste para gerar um resultado falso para as condições `JþK< QUEST`.

Embora o critério de cobertura da condição apareça, à primeira vista, para satisfazer o critério de cobertura da decisão, nem sempre o faz. Se o deci- sion `IF(A & B)` estiver a ser testado, o critério de cobertura da condição permitiria escrever dois casos de teste A é verdadeiro, `B` é falso, e `A` é falso, `B` é verdadeiro - mas isto não faria com que a cláusula `THEN` do `IF fosse` executada. Os testes de cobertura da condição para o exemplo anterior cobriam todos os resultados da decisão, mas isto era apenas por acaso. Por exemplo, dois casos de teste alternativos

```
X¼1, A¼2, B¼1
X¼3, A¼1, X¼1
```

cobrem todos os resultados da condição mas apenas dois dos quatro resultados da decisão (ambos cobrem o caminho *abe* e, portanto, não exercem o verdadeiro resultado da primeira decisão e o falso resultado da segunda decisão).

A forma óbvia de sair deste dilema é um critério denominado cobertura de decisão/condição. Exige casos de teste suficientes, de modo a que cada condição de uma decisão tome todos os resultados possíveis pelo menos uma vez, cada decisão tome todos os resultados possíveis pelo menos uma vez, e cada ponto de entrada seja invocado pelo menos uma vez.

Uma fraqueza com a cobertura de decisão/condição é que, embora possa ap-preparar exercer todos os resultados de todas as condições, frequentemente não o faz, porque certas condições mascaram outras condições. Para ver isto, examinar a Figura 4.2. O fluxograma nesta figura é a forma como um compilador geraria o código da máquina para o programa na Figura 4.1. Os deci- sões multicondições no programa fonte foram quebrados em decisões e ramos individuais porque a maioria

das máquinas não tem uma única instrução que tome decisões de multicondições. Uma cobertura de teste mais completa, então,

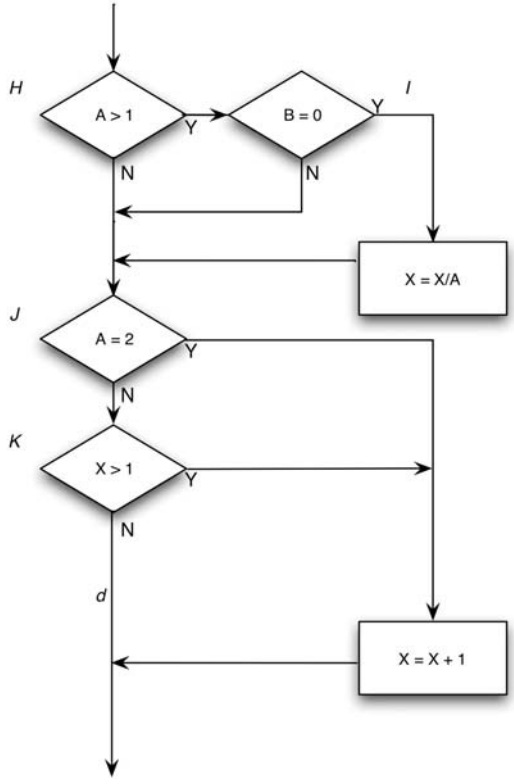


FIGURA 4.2 Código da Máquina para o Programa na Figura 4.1.

parece ser o exercício de todos os resultados possíveis de cada decisão primitiva. Os dois casos de teste de cobertura de decisão anteriores não o conseguem; não conseguem exercer o falso resultado da decisão H e o verdadeiro resultado da decisão K.

A razão, como mostrado na Figura 4.2, é que os resultados das condições no e e o ou expressões podem mascarar ou bloquear a avaliação de outras con- dições. Por exemplo, se uma e condição for falsa, nenhuma das condições subsequentes na expressão precisa de ser avaliada. Do mesmo modo, se uma ou condi- ção for verdadeira, nenhuma das condições subsequentes precisa de ser avaliada. Assim, os erros nas expressões lógicas não são necessariamente revelados pelos critérios de cobertura da condição e de decisão/condição de cobertura.

Um critério que cobre este problema, e depois alguns, é a cobertura de múltiplas condições. Este critério exige que se escrevam casos de teste suficientes, de modo a que todas as combinações possíveis de condições

resultem em cada decisão, e todas

pontos de entrada, são invocados pelo menos uma vez. Por exemplo, considere a sequência de seguimento de pseudo-código.

```
NOTFOUND¼TRUE;
FAÇA I¼1 para TABSIZAR QUANDO (NOTFOUND); /*MESA DE PESQUISA*/
    ... . lógica de busca... ;
FIM
```

As quatro situações a serem testadas são:

1. `I<¼TABSIZE` e `NOTFOUND` é verdade.
2. `I<¼TABSIZE` e `NOTFOUND` é falso (encontrar a entrada antes de atingir o fim da tabela).
3. `I> TABSIZE` e `NOTFOUND` é verdade (acertar no fim da tabela sem encontrar a entrada).
4. `I> TABSIZE` e `NOTFOUND` é falso (a entrada é a última na tabela).

Deve ser fácil ver que um conjunto de casos de teste que satisfazem o critério de condições múltiplas também satisfaz os critérios de cobertura de decisão, cobertura de condição, e cobertura de decisão/condição.

Voltando à Figura 4.1, os casos de teste devem cobrir oito combinações:

1. `A> 1, B¼0`	5. `A¼2, X> 1`
2. `A> 1, B<> 0`	6. `A¼2, X<¼1`
3. `A<¼1, B¼0`	7. `A<> 2, X> 1`
4. `A<¼1, B<> 0`	8. `A<> 2, X<¼1`

Nota Relembrar do trecho de código Java apresentado anteriormente que os casos de teste 5 a 8 valores expressos no ponto do segundo *se* declaração. Uma vez que *X* pode ser alterado acima deste *se* declaração, os valores necessários a este *se* estado- ment devem ser apoiados através da lógica para encontrar os valores de entrada correspondentes.

Estas combinações a serem testadas não implicam necessariamente que sejam necessários oito casos de teste. De facto, podem ser cobertos por quatro casos de teste. Os valores de entrada dos casos de teste, e as combinações que cobrem, são os seguintes:

`A¼2, B¼0, X¼4`	Cobre 1, 5
`A¼2, B¼1, X¼1`	Cobre 2, 6

```
A¼1, B¼0, X¼2      Cobre 3, 7
A¼1, B¼1, X¼1      Cobre 4, 8
```

O facto de existirem quatro casos de teste e quatro caminhos distintos na Figura 4.1 é apenas coincidência. De facto, estes quatro casos de teste não cobrem todos os caminhos; eles falham o caminho *acd*. Por exemplo, seriam necessários oito casos de teste para a seguinte decisão:

```
if(x¼¼y && length(z)¼¼0 && FLAG) {
        j¼1;
senã
o       i¼1;


}
```

embora contenha apenas dois caminhos. No caso de loops, o número de casos de teste exigidos pelo critério de condição múltipla é normalmente muito inferior ao número de trajectórias.

Em resumo, para programas contendo apenas uma condição por decisão, um critério de teste mínimo é um número suficiente de casos de teste para: (1) invocar todos os resultados de cada decisão pelo menos uma vez, e (2) invocar cada ponto de entrada (como ponto de entrada ou *unidade de* entrada) pelo menos uma vez, para assegurar que todos os casos de teste do estado sejam executados pelo menos uma vez. Para programas que contenham decisões com condições múltiplas, o critério mínimo é um número suficiente de casos de teste para invocar todas as combinações possíveis de resultados de condições em cada decisão, e todos os pontos de entrada no programa, pelo menos uma vez. (A palavra "possível" é inserida porque algumas combinações podem ser consideradas impossíveis de criar).

Teste de Black-Box

Tal como discutimos no Capítulo 2, os testes de caixa negra (com base em dados ou com base em entradas/saídas) baseiam-se nas especificações do programa. O objectivo é encontrar áreas em que o programa não se comporte de acordo com as suas especificações.

Particionamento de Equivalência

O capítulo 2 descreveu um bom caso de teste como tendo uma probabilidade- razoável de encontrar um erro; também declarou que é

impossível um teste exaustivo de entrada de uma grama a favor. Assim, ao testar um programa, está limitado a um

pequeno subconjunto de todas as entradas possíveis. É claro, então, que pretende seleccionar o subconjunto "certo", ou seja, o subconjunto com a maior probabilidade de encontrar o maior número de erros.

Uma forma de localizar este subconjunto é perceber que um caso de teste bem seleccionado também deve ter duas outras propriedades:

1. Reduz, em mais do que uma contagem de um, o número de outros casos de teste que devem ser desenvolvidos para atingir algum objectivo pré-definido de testes "rea- sonáveis".
2. Cobre um grande conjunto de outros casos de teste possíveis. Ou seja, diz-nos algo sobre a presença ou ausência de erros para além deste conjunto específico de valores de entrada.

Estas propriedades, embora pareçam ser semelhantes, descrevem duas considerações des- tinctas. A primeira implica que cada caso de teste deve invocar o maior número possível de considerações de entrada diferentes para minimizar o número total de casos de teste necessários. A segunda implica que se deve tentar dividir o domínio de entrada de um programa num número finito de classes de equivalência, de modo a que se possa razoavelmente presumir (mas, claro, não ter a certeza absoluta) que um teste de um valor representativo de cada classe é equivalente a um teste de qualquer outro valor. Ou seja, se um caso de teste numa classe de equivalência detectar um erro, esperar-se-ia que todos os outros casos de teste na classe de equivalência encontrassem o mesmo erro. Pelo contrário, se um caso de teste não detectasse um erro, seria de esperar que nenhum outro caso de teste na classe de equivalência se inserisse noutra classe de equivalência, uma vez que as classes de equivalência podem sobrepor-se umas às outras.

Estas duas considerações formam uma metodologia de caixa negra conhecida como partição por equivalência. A segunda consideração é utilizada para desenvolver um conjunto de condições "interessantes" a serem testadas. A primeira consideração é então utilizada para desenvolver um conjunto mínimo de casos de teste abrangendo estas condições.

Um exemplo de uma classe de equivalência no programa triangular do Capítulo 1 é o conjunto "três números de igual valor com valores inteiros superiores a zero". Ao identificar isto como uma classe de equivalência, estamos a afirmar que se nenhum erro for encontrado por um teste de um elemento do conjunto, é improvável que um erro seja encontrado por um teste de outro elemento do conjunto. Por outras palavras, o nosso tempo de teste é melhor gasto noutro local: em diferentes classes de equivalência.

O desenho da caixa de ensaio por equivalência de partição prossegue em duas etapas:

(1) identificação das classes de equivalência e (2) definição dos casos de teste.

Exterior condição	Equivalência válida aulas	Equivalência inválida aulas

FIGURA 4.3 Um Formulário para Enumerar Classes de Equivalência.

Identificar as Classes de Equivalência As classes de equivalência são definidas tomando cada condição de entrada (geralmente uma frase ou frase na especificação) e dividindo-a em dois ou mais grupos. Pode utilizar a tabela da Figura 4.3 para o fazer. Note que são identificados dois tipos de classes de equivalência: as classes de equivalência válidas representam entradas válidas para a pró-grama, e as classes de equivalência inválidas representam todos os outros estados possíveis da condição (ou seja, valores de entrada errados). Assim, estamos a aderir ao princípio 5, discutido no Capítulo 2, que afirma que se deve concentrar a atenção nas condições inválidas ou inesperadas.

Dado um input ou condição externa, a identificação da classe de equivalência - ses é em grande parte um processo heurístico. Siga estas directrizes:

1. Se uma condição de entrada especificar um intervalo de valores (por exemplo, "a contagem de itens pode ser de 1 a 999"), identificar uma classe de equivalência válida (`1< contagem de itens< 999`) e duas classes de equivalência inválidas (`contagem de itens< 1 e contagem de itens> 999`).
2. Se uma condição de entrada especificar o número de valores (por exemplo, "um a seis proprietários podem ser listados para o automóvel"), identifique uma classe de equivalência válida e duas classes de

equivalência inválidas (nenhum proprietário e mais de seis proprietários).

3. Se uma condição de entrada especifica um conjunto de valores de entrada, e há razões para acreditar que o programa trata cada um de forma diferente ("tipo

de veículo deve ser BUS, TRUCK, TAXICAB, PASSENGER, ou MOTORCYCLE"), identificar uma classe de equivalência válida para cada e uma classe de equivalência inválida ("TRAILER," por exemplo).
4. Se uma condição de entrada especificar uma situação de "deve ser", tal como "o primeiro carácter do identificador deve ser uma letra", identificar uma classe de equivalência válida (é uma letra) e uma classe de equivalência inválida (não é uma letra).

Se houver alguma razão para acreditar que o programa não trata de forma idêntica os elementos de uma classe de equivalência, dividir a classe de equivalência em classes de equivalência mais pequenas. Em breve ilustraremos um exemplo deste processo.

Identificação dos casos de teste A segunda etapa é a utilização de classes de equivalência para identificar os casos de teste. O processo é o seguinte:

1. Atribuir um número único a cada classe de equivalência.
2. Até que todas as classes de equivalência válidas tenham sido cobertas por casos de teste (com classificação incorpórea), escrever um novo caso de teste cobrindo o maior número possível das classes de equivalência válidas não cobertas.
3. Até que os seus casos de teste tenham coberto todas as classes de equivalência inválidas, escreva um caso de teste que cubra uma, e apenas uma, das classes de equivalência inválidas não cobertas.

A razão pela qual os casos de teste individuais cobrem casos inválidos é que certos controlos de erros de entrada mascaram ou substituíram outros controlos de erros de entrada. Por exemplo, se a especificação indicar "introduzir tipo de livro (HARDCOVER, SOFTCOVER, ou LOOSE) e montante (1-999)', o caso teste, (`XYZ 0`), expressando duas condições de erro (tipo de livro inválido e montante), provavelmente não exercerá a verificação do montante, uma vez que o programa pode dizer "XYZ É TIPO DE LIVRO INCORRIDO" e não se preocupar em examinar o resto - der da entrada.

Um exemplo

Como exemplo, assumimos que estamos a desenvolver um compilador para um subconjunto da linguagem Fortran, e desejamos testar a verificação da sintaxe da declaração `DIMENSION`. A especificação é

listada abaixo. (Nota: Isto não é

a declaração completa da Fortran *DIMENSION*; foi editada consideravelmente para a tornar em tamanho de livro de texto. Não se iluda ao pensar que o teste de programas reais é tão fácil como os exemplos deste livro). No específico, os itens em itálico indicam unidades sintácticas para as quais entidades específicas devem ser substituídas em declarações reais; são utilizados parênteses para indicar itens op- tion; e uma elipse indica que o item anterior pode aparecer várias vezes sucessivamente.

> Uma declaração de *DIMENSÃO* é utilizada para especificar as dimensões das matrizes. A forma da declaração de *DIMENSÃO* é
>
> DIMENSÃO *ad*[,ad]...
>
> onde o anúncio é um descritor de matriz da forma
>
> n(d[,d]...)
>
> onde n é o nome simbólico da matriz e d é um declarador de dimensão. Os nomes simbólicos podem ser de uma a seis letras ou dígitos, o primeiro dos quais deve ser uma letra. Os números mínimo e máximo das declarações de dimensão que podem ser especificados para uma matriz são um e sete, respectivamente. A forma de um declarador de dimensão é
>
> [lb:]ub
>
> onde lb e ub são os limites das dimensões inferior e superior. Um limite pode ser uma constante no intervalo −65534 a 65535 ou o nome de uma variável inteira (mas não o nome de um elemento de matriz). Se lb não for especificado, assume-se que é 1. O valor de ub deve ser maior ou igual a lb. Se lb for especificado, o seu valor pode ser negativo, 0, ou posi- tive. Quanto a todas as declarações, a declaração de *DIMENSÃO* pode ser contínua - em várias linhas.

O primeiro passo consiste em identificar as condições de entrada e, a partir delas, localizar as classes de equivalência. Estas são tabuladas na Tabela 4.1. Os números na tabela são identificadores únicos das classes de equivalência.

O passo seguinte é escrever um caso de teste cobrindo uma ou mais classes de equívocos válidos. Por exemplo, o caso de teste

DIMENSÃO A(2)

cobre as classes 1, 4, 7, 10, 12, 15, 24, 28, 29, e 43.

TABELA 4.1 Classes de Equivalência

Condição de entrada	Equivalência válida Aulas	Equivalência Inválida Aulas
Número de descritores de matriz	um (1), > um (2)	nenhum (3)
Tamanho do nome da matriz	1–6 (4)	0 (5), > 6 (6)
Nome da matriz	tem letras (7), tem dígitos (8)	tem outra coisa (9)
O nome da matriz começa com letteryes (10)		no (11)
Número de dimensões	1–7 (12)	0 (13), > 7 (14)
O limite superior é	constante (15), variável inteira (16)	nome do elemento da matriz (17), algo mais (18)
A letra de identificação variável inteira (19), tem dígitos (20)		tem outra coisa (21)
A variável inteira começa com a letra	sim (22)	não (23)
Constante	–65534–65535 (24)	< –65534 (25), > 65535 (26)
Limite inferior especificado (28)	sim (27), não	
Limite superior a limite inferior	superior a (29), igual (30)	menos de (31)
Limite inferior negativo especificado (32), zero (33), > 0 (34)		
O limite inferior é	constante (35), variável inteira (36)	nome do elemento da matriz (37), algo mais (38)
O limite inferior é	um (39)	ub>¼1 (40), ub< 1 (41)
Múltiplas linhas	sim (42), não (43)	

O passo seguinte é conceber um ou mais casos de teste que cubram as restantes classes de equivalência válidas. Um caso de teste do formulário

```
DIMENSÃO A 12345 (I,9,J4XXXX,65535,1,KLM,
```

```
X,1000, BBB(-65534:100,0:1000,10:10, I:65535)
```

cobre as restantes classes. As classes de equivalência de entradas inválidas, e um caso de teste representando cada uma delas, são:

```
  (3): DIMENSÃO
  (5): DIMENSÃO (10)
  (6): DIMENSÃO A234567(2)
  (9): DIMENSÃO A.1(2)
(11): DIMENSÃO 1A(10)
(13): DIMENSÃO B
(14): DIMENSÃO B(4,4,4,4,4,4,4,4,4)
(17): DIMENSÃO B(4,A(2))
(18): DIMENSÃO B(4,,7)
(21): DIMENSÃO C(I.,10)
(23): DIMENSÃO C(10,1J)
(25): DIMENSÃO D(- 65535:1)
(26): DIMENSÃO D(65536)
(31): DIMENSÃO D(4:3)
(37): DIMENSÃO D(A(2):4)
(38): D(.:4)
(43): DIMENSÃO D(0)
```

Assim, as classes de equivalência foram abrangidas por 17 casos de teste. Poderá querer considerar como estes casos de teste se compararam a um conjunto de casos de teste derivados de uma forma ad hoc.

Embora a partição por equivalência seja muito superior a uma selecção aleatória de casos de teste, ainda apresenta deficiências. Esquece, por exemplo, certos tipos de casos de teste de alto rendimento. As duas metodologias seguintes, análise de valores-limite e gráficos de causa-efeito, cobrem muitas destas deficiências.

Análise do Valor Limite

A experiência mostra que os casos de teste que exploram condições-limite têm um retorno mais elevado do que os casos de teste que não o fazem. As condições-limite são as situações directamente sobre, acima e abaixo dos limites das classes de equivalência de entrada e das classes de equivalência de saída. A análise do valor-limite difere da partição da equivalência em dois aspectos:

1. Em vez de seleccionar qualquer elemento de uma classe de equivalência como sendo representativo, a análise do valor limite exige que um ou mais elementos sejam seleccionados de modo a que cada margem da classe de equivalência seja objecto de um teste.

2. Em vez de focar apenas a atenção nas condições de entrada (espaço de entrada), os casos de teste são também derivados considerando o espaço de resultados (classes de equivalência de saída).

É difícil apresentar um "livro de cozinha" para a análise de valores-limite, uma vez que requer um grau de criatividade e uma certa especialização em relação ao problema em questão. (Assim, como muitos outros aspectos dos testes, é mais um estado de espírito do que qualquer outra coisa). No entanto, algumas linhas de orientação gerais estão em ordem:

1. Se uma condição de entrada especifica uma gama de valores, escrever casos de teste para os fins da gama, e casos de teste de entrada inválida para situações imediatamente para além dos fins. Por exemplo, se o domínio válido de um valor de entrada for -1,0 a 1,0, escrever casos de teste para as situações -1,0, 1,0, -1,001,
   e 1.001.
2. Se uma condição de entrada especificar um número de valores, escrever casos de teste para o número mínimo e máximo de valores e um abaixo e além desses valores. Por exemplo, se um ficheiro de entrada pode conter 1-255 registos, escrever casos de teste para 0, 1, 255, e 256 registos.
3. Utilizar a directriz 1 para cada condição de saída. Por exemplo, se um programa de folha de pagamento calcular a dedução mensal do FICA, e se o mínimo for $0,00 e o máximo for $1,165,25, escrever casos de teste que causem
   $0,00 e $1,165,25 a deduzir. Ver também se é possível inventar casos de teste que possam causar uma dedução negativa ou uma dedução de mais de $1.165,25.

   Note-se que é importante examinar os limites do espaço de resultados porque nem sempre os limites dos domínios in- put representam o mesmo conjunto de circunstâncias que os limites das faixas de saída (por exemplo, considere uma sub-rotina senoidal). Além disso, nem sempre é possível gerar um resultado fora do intervalo de saída; no entanto, vale a pena considerar a possibilidade.
4. Utilizar a linha directriz 2 para cada condição de saída. Se um sistema de recuperação de informação exibir os resumos mais relevantes com base numa nova busca de entrada, mas nunca mais do que quatro resumos, escrever casos de teste tais que o programa exiba zero, um, e quatro resumos, e escrever um caso de teste que possa fazer com que o programa exiba erroneamente cinco resumos.
5. Se a entrada ou saída de um programa for um conjunto ordenado (um ficheiro sequencial, por exemplo, ou uma lista linear ou uma tabela), concentre a atenção no primeiro e último elementos do conjunto.

6. Além disso, use o seu engenho para procurar outras condições de fronteira.

O programa de análise triangular do Capítulo 1 pode ilustrar a necessidade de análise de valores-limite. Para que os valores de entrada representem um triângulo, devem ser inteiros superiores a 0 onde a soma de quaisquer dois é superior ao terceiro. Se estivesse a definir partições equivalentes, poderia definir uma em que esta condição fosse satisfeita e outra em que a soma de dois dos números inteiros não fosse maior do que o terceiro. Assim, dois casos de teste possíveis podem ser 3-4-5 e 1-2-4. No entanto, falhámos um erro provável. Ou seja, se uma expressão no programa fosse codificada como AþB>¼C em vez de AþB> C, o programa dir-nos-ia erroneamente que 1-2-3 representa um triângulo escaleno válido. Assim, a diferença importante entre a análise de valores-limite e a partição de equivalência é que a análise de valores-limite explora situ- ations sobre e em torno das bordas das partições de equivalência.

Como exemplo de uma análise de valores-limite, considere a seguinte especificação do programa:

> O MTEST é um programa que classifica os exames de escolha múltipla. A entrada é um ficheiro de dados chamado OCR, com registos múltiplos com 80 caracteres. Segundo a especificação do ficheiro, o primeiro registo é um título utilizado como título em cada relatório de saída. O próximo conjunto de registos descreve as respostas correctas no exame. Estes registos contêm um "2" como o último caracter na coluna 80. No primeiro registo deste conjunto, o número de perguntas é listado nas colunas 1-3 (um valor de 1-999). As colunas 10-59 contêm as respostas correctas para as perguntas 1-50 (qualquer carácter é válido como resposta). Os registos subsequentes contêm, nas colunas 10-59, as respostas correctas para as perguntas 51-100, 101-150, e assim por diante.
>
> O terceiro conjunto de registos descreve as respostas de cada estudante; cada um destes registos contém um "3" na coluna 80. Para cada aluno, o primeiro registo contém o nome ou número do aluno nas colunas 1- 9 (quaisquer caracteres); as colunas 10-59 contêm as respostas do aluno às perguntas 1-50. Se o teste tiver mais de 50 perguntas, os registos subsequentes para o aluno contêm as respostas 51-100, 101-150, e assim por diante, nas colunas 10-59. O número máximo de alunos é de 200. Os dados de entrada são ilustrados na Figura 4.4. Os quatro registos de saída são:
>
> 1. Um relatório, classificado por identificador de estudante, mostrando a nota (percentagem de respostas correctas) e a classificação de cada estudante.
> 2. Um relatório semelhante, mas ordenado por nota.

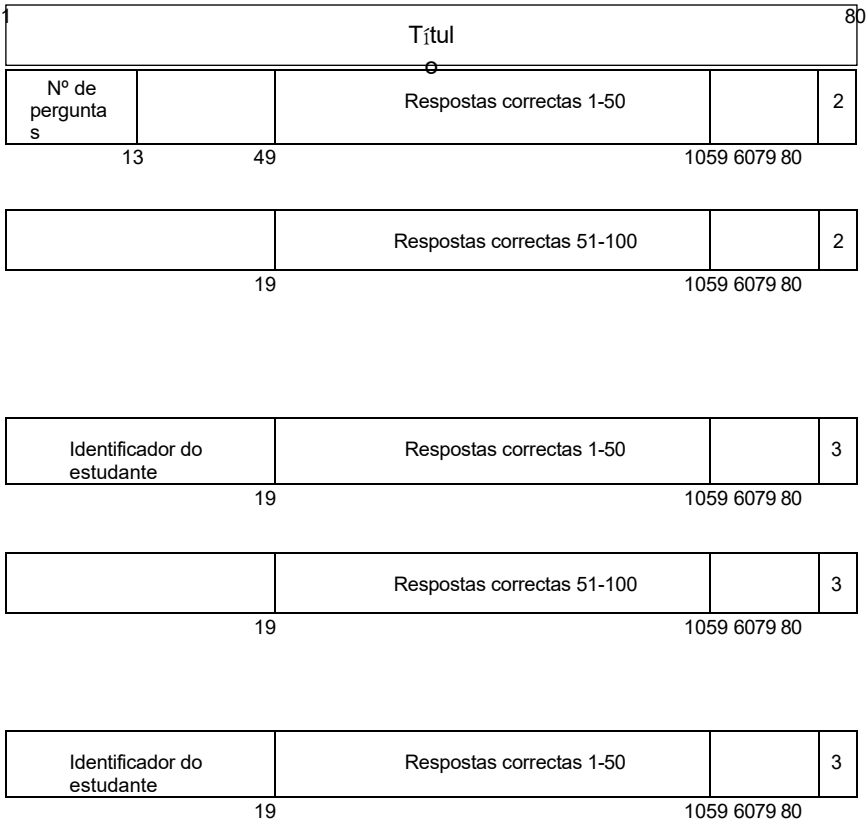


FIGURA 4.4 Entrada para o Programa MTEST.

3. Um relatório indicando a média, a mediana e o desvio padrão das notas.
4. Um relatório, ordenado por número de pergunta, mostrando a idade percentual dos estudantes que respondem correctamente a cada pergunta.

Podemos começar por ler metodicamente a especificação, procurando as condições no terreno. A primeira condição de entrada de limite é um ficheiro de entrada vazio. A segunda condição de entrada é o registo de título; as condições de limite são um registo de título em falta e os títulos mais curtos e mais longos possíveis. As condições de entrada seguintes são a presença de registos de resposta correcta e o campo do número de perguntas no primeiro registo de resposta. A classe de equivalência

para o número de perguntas não é 1-999, porque algo especial de hapcanetas em cada múltiplo de 50 (ou seja, são necessários múltiplos registos). Uma razão- capaz de dividir isto em classes de equivalência é 1-50 e 51-999. Assim, precisamos de casos de teste em que o campo do número de perguntas é fixado em 0, 1, 50, 51, e 999. Isto abrange a maior parte das condições limite para o número de registos de respostas correctas; contudo, mais três situações interessantes são a ausência de registos de respostas e o número de registos de respostas a mais e a menos (por exemplo, o número de perguntas é 60, mas há três registos de respostas num caso e um registo de respostas no outro caso). Os casos de teste únicos identificados até agora são:

1. Ficheiro de entrada vazio
2. Registo de título em falta
3. Título de 1-caractere
4. Título com 80 caracteres
5. 1-exame de perguntas
6. Exame de 50-questões
7. 51-exame de perguntas
8. 999-exame de perguntas
9. Exame 0-question
10. Campo de número de perguntas com valor não-numérico
11. Sem registos de resposta correcta após o registo do título
12. Um número demasiado elevado de registos de respostas correctas
13. Um número demasiado reduzido de registos de respostas correctas

As condições de entrada seguintes estão relacionadas com as respostas dos estudantes. Os casos de teste de valor limite aqui parecem ser:

14. 0 estudantes
15. 1 estudante
16. 200 estudantes
17. 201 estudantes
18. Um estudante tem um registo de resposta, mas existem dois registos de resposta correcta.
19. O aluno acima é o primeiro aluno do ficheiro.
20. O aluno acima é o último aluno no ficheiro.
21. Um estudante tem dois registos de respostas, mas há apenas um registo de respostas correctas.

22. O aluno acima é o primeiro aluno do ficheiro.
23. O aluno acima é o último aluno no ficheiro.

Também se pode derivar um conjunto útil de casos de teste examinando os limites de saída, embora alguns dos limites de saída (por exemplo, relatório 1 vazio) sejam cobertos pelos casos de teste existentes. As condições de limite dos relatórios 1 e 2 são:

0 alunos (o mesmo que o teste 14)
1 aluno (o mesmo que o teste 15)
200 estudantes (o mesmo que o teste 16)

24. Todos os estudantes recebem a mesma nota.
25. Todos os estudantes recebem uma nota diferente.
26. Alguns, mas não todos, os estudantes recebem a mesma nota (para ver se as classificações são computadas correctamente).
27. Um estudante recebe uma nota de 0.
28. Um estudante recebe uma nota de 10.
29. Um estudante tem o menor valor identificador possível (para verificar o tipo).
30. Um estudante tem o maior valor identificador possível.
31. O número de estudantes é tal que o relatório é apenas suficientemente grande para caber numa página (para ver se uma página estranha é impressa).
32. O número de estudantes é tal que todos os estudantes, excepto um, cabem numa página.

As condições limite do relatório 3 (média, mediana e desvio padrão) são:

33. A média está no seu máximo (todos os estudantes têm uma pontuação perfeita).
34. A média é 0 (todos os estudantes recebem uma nota de 0).
35. O desvio padrão está no seu máximo (um aluno recebe um 0 e o outro um 100).
36. O desvio padrão é 0 (todos os estudantes recebem a mesma nota).

Os testes 33 e 34 também cobrem os limites da mediana. Outro caso de teste útil é a situação em que há 0 alunos (procurando uma divisão por 0 no cálculo da média), mas isto é idêntico ao caso de teste 14.

Um exame do relatório 4 produz os seguintes testes de valores-limite:

37. Todos os estudantes respondem correctamente à pergunta 1.
38. Todos os estudantes respondem incorrectamente à pergunta 1.
39. Todos os estudantes respondem correctamente à última pergunta.
40. Todos os estudantes respondem incorrectamente à última pergunta.
41. O número de perguntas é tal que o relatório é apenas suficientemente grande para caber numa página.
42. O número de perguntas é tal que todas as perguntas, excepto uma, cabem numa página.

Um programador experiente provavelmente concordaria nesta altura que muitos destes 42 casos de teste representam erros comuns que poderiam ter sido cometidos no desenvolvimento deste programa, mas a maioria destes erros provavelmente não seriam detectados se fosse utilizado um método de geração de casos de teste aleatórios ou ad hoc. A análise do valor-limite, se praticada correctamente, é um dos métodos mais úteis para a concepção de casos de teste. Contudo, é frequentemente utilizada de forma ineficaz, porque a técnica, na superfície, soa simples. Deve compreender que as condições-limite podem ser muito subtis e, portanto, a identificação das mesmas requer muita reflexão.

Gráfico de Causa-Efeito

Um ponto fraco da análise de valores-limite e da partição de equivalência é que não exploram combinações de circunstâncias de entrada. Por exemplo, talvez o programa MTEST da secção anterior falhe quando a prova do número de perguntas e o número de estudantes ultrapassa algum limite (o programa fica sem memória, por exemplo). O teste do valor limite não detectaria necessariamente um tal erro.

O teste das combinações de entrada não é uma tarefa simples, porque mesmo que se partilhe as condições de entrada por equivalência, o número de combinações é geralmente astronómico. Se não tiver uma forma sistemática de seleccionar um subconjunto de condições de entrada, provavelmente seleccionará um subconjunto arbitrário de condições, o que poderá levar a um teste ineficaz.

Os gráficos de causa e efeito ajudam a seleccionar, de forma sistemática, um conjunto de casos de teste de alto rendimento. Tem um efeito secundário benéfico ao apontar ambiguidades e nessidades incompletas na especificação.

Um gráfico de causa-efeito é uma linguagem formal para a qual é traduzida uma especificação de linguagem natural. O gráfico é na realidade um circuito lógico digital (uma rede lógica combinatória), mas em vez da notação electrónica padrão, é utilizada uma notação um pouco mais simples. Nenhum conhecimento de electrónica é necessário - essencial para além de uma compreensão da lógica booleana (ou seja, da operação lógica - atores e, ou, e não).

O processo seguinte é utilizado para derivar casos de teste:

1. A especificação está dividida em peças funcionais. Isto é necessário porque o gráfico de causa-efeito se torna pesado quando usado em grandes especulações. Por exemplo, ao testar um sistema de comércio electrónico, uma peça funcional pode ser a especificação para a escolha e verificação de um único artigo colocado num carrinho de compras. Ao testar um desenho de página Web, pode testar uma única árvore de menu ou mesmo uma sequência de navegação menos complexa.
2. As causas e os efeitos na especificação são identificados. Uma causa é uma condição de entrada distinta ou uma classe de equivalência de condições de entrada. Um efeito é uma condição de saída ou uma transformação do sistema (um efeito de permanência que uma entrada tem sobre o estado do programa ou sistema). Por exemplo, se uma transacção causar a actualização de um ficheiro ou de um registo de base de dados, a alteração é uma transformação do sistema; uma mensagem de confirmação seria uma condição de saída.

   Identifica causas e efeitos lendo a especificação palavra por palavra e sublinhando palavras ou frases que descrevem causas e efeitos. Uma vez identificada, a cada causa e efeito é atribuído um número único.
3. O conteúdo semântico da especificação é analisado e trans-formado num gráfico booleano ligando as causas e efeitos. Este é o gráfico de causa-efeito.
4. O gráfico é anotado com restrições descrevendo combinações de causas e/ou efeitos que são impossíveis devido a restrições sintácticas ou ambientais.
5. Ao traçar metodicamente as condições de estado no gráfico, converte-se o gráfico numa tabela de decisão de entrada limitada. Cada coluna na tabela representa um caso de teste.
6. As colunas na tabela de decisão são convertidas em casos de teste.

A notação básica para o gráfico é mostrada na Figura 4.5. Pense em cada

nó como tendo o valor 0 ou 1; 0 representa o estado 'ausente'e 1 repreende o estado "presente".

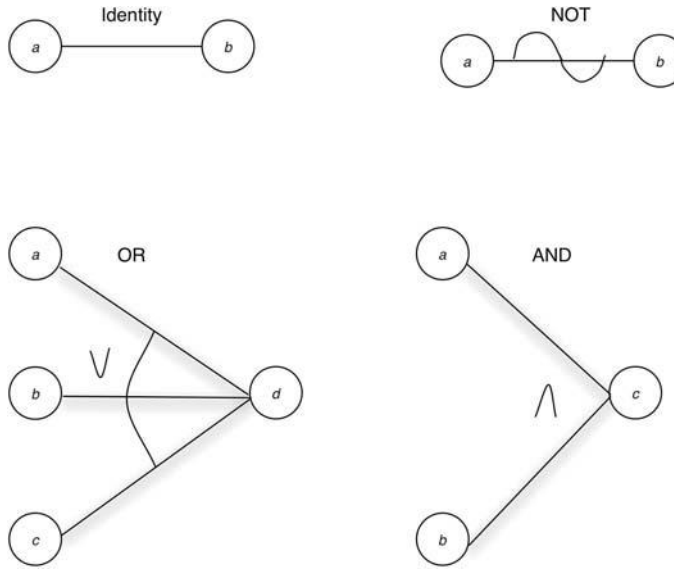


FIGURA 4.5 Símbolos Básicos do Gráfico de Causa-Efeito.

- A função de identidade indica que se a é 1, b é 1; caso contrário, b é 0.
- A função não indica que se a é 1, b é 0, senão b é 1.
- A ou função declara que se a ou b ou c é 1, d é 1; caso contrário d é 0.
- O e função afirma que se tanto a como b são 1, c é 1; caso contrário, c é 0.

As duas últimas funções (ou e e) podem ter qualquer número de entradas.

Para ilustrar um pequeno gráfico, considere a seguinte especificação:

O personagem da coluna 1 deve ser um "A" ou um "B". O carácter na coluna 2 tem de ser um dígito. Nesta situação, é feita a actualização do ficheiro. Se o primeiro caractere estiver incorrecto, é emitida a mensagem `X12`. Se o segundo caractere não for um dígito, é emitida a mensagem `X13`.

As causas são:

1-caracter na coluna 1 é "A" 2-caracter na coluna 1 é "B" 3-caracter na coluna 2 é um

dígito

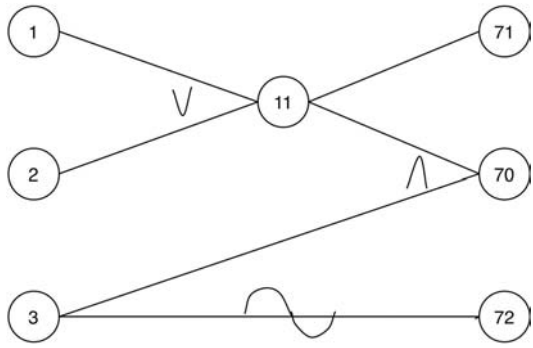


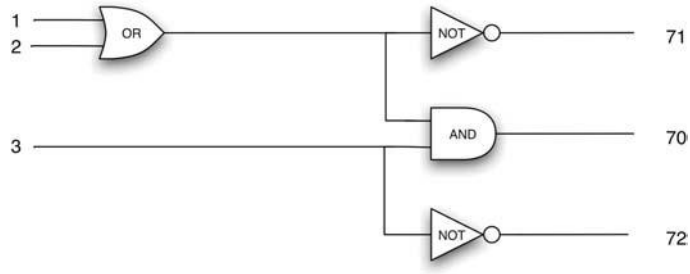
FIGURA 4.6 Gráfico de Causa-Efeito da Amostra.

e os efeitos são:

70 actualizações feitas 71-
mensagem `X12` é emitida
72-mensagem `X13` é
emitida

O gráfico de causa-efeito é mostrado na Figura 4.6. Note-se o nó intermédio 11 que foi criado. Deve confirmar que o gráfico representa a especificação, definindo todos os estados possíveis das causas e verificando que os efeitos são definidos para os valores correctos. Para os leitores familiarizados com a lógica dia- gramas, a Figura 4.7 é o circuito lógico equivalente.

Embora o gráfico da Figura 4.6 represente a especificação, contém uma combinação impossível de causas - é impossível que ambas as causas 1 e 2 sejam definidas para 1 simultaneamente. Na maioria dos programas, certas combinações de causas são impossíveis devido a considerações sintácticas ou ambientais (um personagem não pode ser um "A" e um "B" simultaneamente).



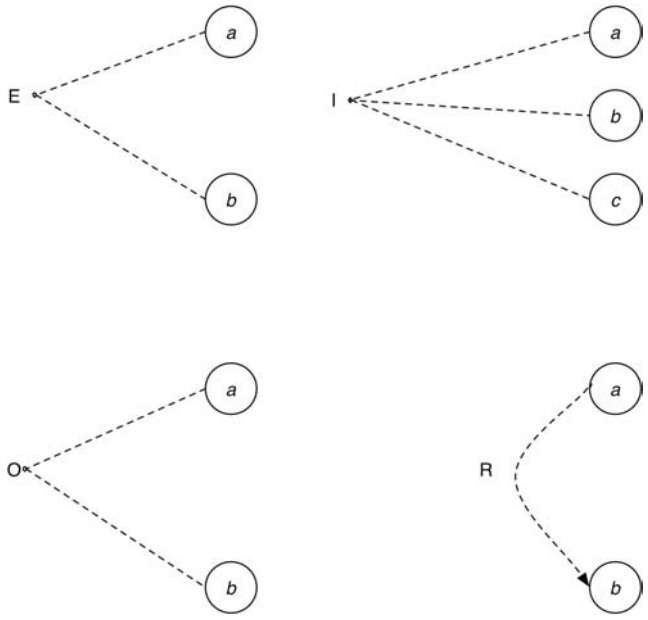


FIGURA 4.8 Símbolos de Restrição.

Para os explicar, é utilizada a notação da Figura 4.8. A restrição E afirma que deve ser sempre verdade que, no máximo, um de a e b pode ser 1 (a e b não podem ser 1 simultaneamente). A restrição I afirma que pelo menos um de a, b, e c deve ser sempre 1 (a, b, e c não pode ser 0 em simultâneo). O constrangimento O declara que um, e apenas um, de a e b tem de ser 1. A restrição R declara que para a ser 1, b deve ser 1 (ou seja, é impossível que a seja 1 e b seja 0).

Existe frequentemente a necessidade de um constrangimento entre os efeitos. O M con- straint na Figura 4.9 afirma que se o efeito a for 1, o efeito b é forçado a 0.

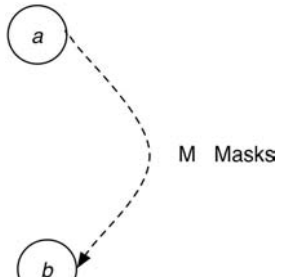


FIGURA 4.9 Símbolo para "Máscaras" Restrições.

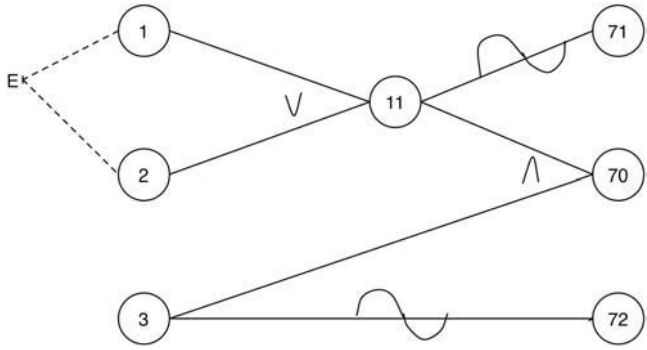


FIGURA 4.10 Gráfico de Causa-Efeito de Amostra com Restrição "Exclusiva".

Voltando ao simples exemplo anterior, vemos que é fisicamente impossível que as causas 1 e 2 estejam presentes simultaneamente, mas é possível - sangrar por nenhuma delas estar presente. Assim, elas estão ligadas à restrição E, como se mostra na Figura 4.10.

Para ilustrar como o gráfico de causa-efeito é utilizado para derivar casos de teste, utilizamos a seguinte especificação para um comando de depuração num sistema interactivo.

> O comando *DISPLAY* é utilizado para visualizar a partir de uma janela terminal o conteúdo dos locais de memória. A sintaxe do comando é mostrada na Figura 4.11. Os parênteses representam operandos opcionais alternativos. As letras maiúsculas representam as palavras-chave dos operandos. As letras minúsculas representam valores de operando (os valores reais devem ser substituídos). Os op- erands sublinhados representam os valores por defeito (ou seja, o valor utilizado quando o op- erand é omitido).

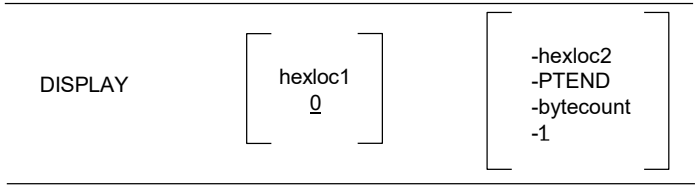


FIGURA 4.11 Sintaxe do Comando DISPLAY.

O primeiro operando (*hexloc1*) especifica o endereço do primeiro byte cujo conteúdo deve ser exibido. O endereço pode ter de um a seis dígitos hexadecimais (0-9, A-F) de comprimento. Se não for especificado, supõe-se que o ad vestido 0. O endereço deve estar dentro da área de memória real da máquina.

O segundo operando especifica a quantidade de memória a ser exibida. Se for especificado *hexloc2*, define o endereço do último byte no intervalo de localizações a ser exibido. Pode ser de um a seis dígitos hexadecimais de comprimento. O endereço deve ser maior ou igual ao endereço inicial (*hexloc1*). Além disso, *o hexloc2* deve estar dentro da gama de memória real da ma-chine. Se `FIM` for especificado, a memória é exibida através do último byte real na máquina. Se for especificada a `contagem bytec,` é desmineralizado o número de bytes de memória a ser exibido (começando com o local especificado em *hexloc1*). O *bytecount* operando é um inteiro hexadecimal (de um a seis dígitos). A soma de `bytecount` e *hexloc1* não deve exceder o tamanho real da memória mais 1, e o *bytecount* deve ter um valor de pelo menos 1.

Quando o conteúdo da memória é apresentado, o formato de saída no ecrã é uma ou mais linhas do formato

```
xxxxxx ¼ palavra1 palavra2 palavra3 palavra4
```

onde `xxxxxx` é o endereço hexadecimal da `palavra1`. É sempre exibido um número integral de palavras (sequências de quatro bytes, em que o endereço do primeiro byte da palavra é um múltiplo de 4), sem garra do valor de *hexadecimal1* ou a quantidade de memória a ser exibida. Todas as linhas de saída conterão sempre quatro palavras (16 bytes). O primeiro byte do intervalo exibido cairá dentro da primeira palavra.

As mensagens de erro que podem ser produzidas são

M1 é uma sintaxe de comando inválida.

A memória M2 solicitada está para além do limite de memória real.

A memória M3 solicitada é um intervalo zero ou negativo.

Como exemplos:

```
DISPLAY
```

exibe as primeiras quatro palavras na memória (endereço inicial padrão de 0, contagem de bytes padrão de 1);

```
DISPLAY 77F
```

exibe a palavra que contém o byte no endereço 77F, e as três palavras subsequentes;

```
VISOR 77F-407A
```

exibe as palavras que contêm os bytes na gama de endereços 775-407A;

```
DISPLAY 77F.6
```

exibe as palavras contendo os seis bytes que começam no local 77F; e

```
MOSTRAR 50FF-END
```

exibe as palavras contendo os bytes na gama de endereços 50FF até ao fim da memória.

O primeiro passo é uma análise cuidadosa da especificação para identificar as causas e efeitos. As causas são as seguintes:

1. O primeiro operando está presente.
2. O operando hexloc1 contém apenas dígitos hexadecimais.
3. O operando hexloc1 contém de um a seis caracteres.
4. O operando hexloc1 está dentro da gama de memória real da máquina.
5. O segundo operando é o *FIM*.
6. O segundo operando é o hexloc.
7. O segundo operando é bytecount.
8. O segundo operando é omitido.
9. O operando hexloc2 contém apenas dígitos hexadecimais.
10. O operando hexloc2 contém de um a seis caracteres.
11. O operando hexloc2 está dentro da gama de memória real da máquina.
12. O operando hexloc2 é maior ou igual ao operando hexloc1.
13. O bytecount operand contém apenas dígitos hexadecimais.
14. O bytecount operand contém de um a seis caracteres.

15. bytecount þhexloc1 <¼ tamanho da memória þ 1.
16. bytecount >¼ 1.
17. A gama especificada é suficientemente grande para requerer múltiplas linhas de saída.
18. O início do alcance não cai sobre um limite de palavras.

A cada causa foi atribuído um número único arbitrário. Note-se que quatro causas (5 a 8) são necessárias para o segundo operando porque o operando sec- ond poderia ser (1) *END*, (2) hexloc2, (3) byte-count, (4) ausente, e
(5) nenhuma das anteriores. Os efeitos são os seguintes:

91. A mensagem `M1` é exibida.
92. A mensagem `M2` é exibida.
93. A mensagem `M3` é exibida.
94. A memória é exibida numa linha.
95. A memória é exibida em várias linhas.
96. O primeiro byte de alcance apresentado cai sobre um limite de palavras.
97. O primeiro byte de alcance exibido não cai sobre um limite de palavras.

O passo seguinte é o desenvolvimento do gráfico. Os nós de causa são listados verticalmente no lado esquerdo da folha de papel; os nós de efeito são listados verticalmente no lado direito. O conteúdo semântico da especificação é cuidadosamente analisado para interligar as causas e efeitos (ou seja, para mostrar em que condições um efeito está presente).

A Figura 4.12 mostra uma versão inicial do gráfico. O nó intermediário 32 representa um primeiro operando sintacticamente válido; o nó 35 representa um segundo operando sintacti- cally válido. O nó 36 representa um com- mand sintacticamente válido. Se o nó 36 for 1, o efeito 91 (a mensagem de erro) não aparece. Se o nó 36 for 0, o efeito 91 está presente.

O gráfico completo é mostrado na Figura 4.13. Deve explorá-lo cuidadosamente para se convencer de que reflecte com precisão a especificação.

Se a Figura 4.13 fosse utilizada para derivar os casos de teste, muitos casos de teste impossíveis de criar seriam derivados. A razão é que certas combinações de causas são impossíveis devido a restrições sintácticas. Por exemplo, as causas 2 e 3 não podem estar presentes, a menos que a causa 1 esteja presente. A causa 4 não pode estar presente, a menos que ambas as

causas 2 e 3 estejam presentes. A Figura 4.14 contém o gráfico completo com as condições de restrição. Note-se que, no máximo, uma das causas 5, 6, 7, e 8 pode estar presente. Todas as outras restrições de causa são os requisitos de con- dição. Note-se que a causa 17 (múltiplas linhas de saída) requer a não causa 8

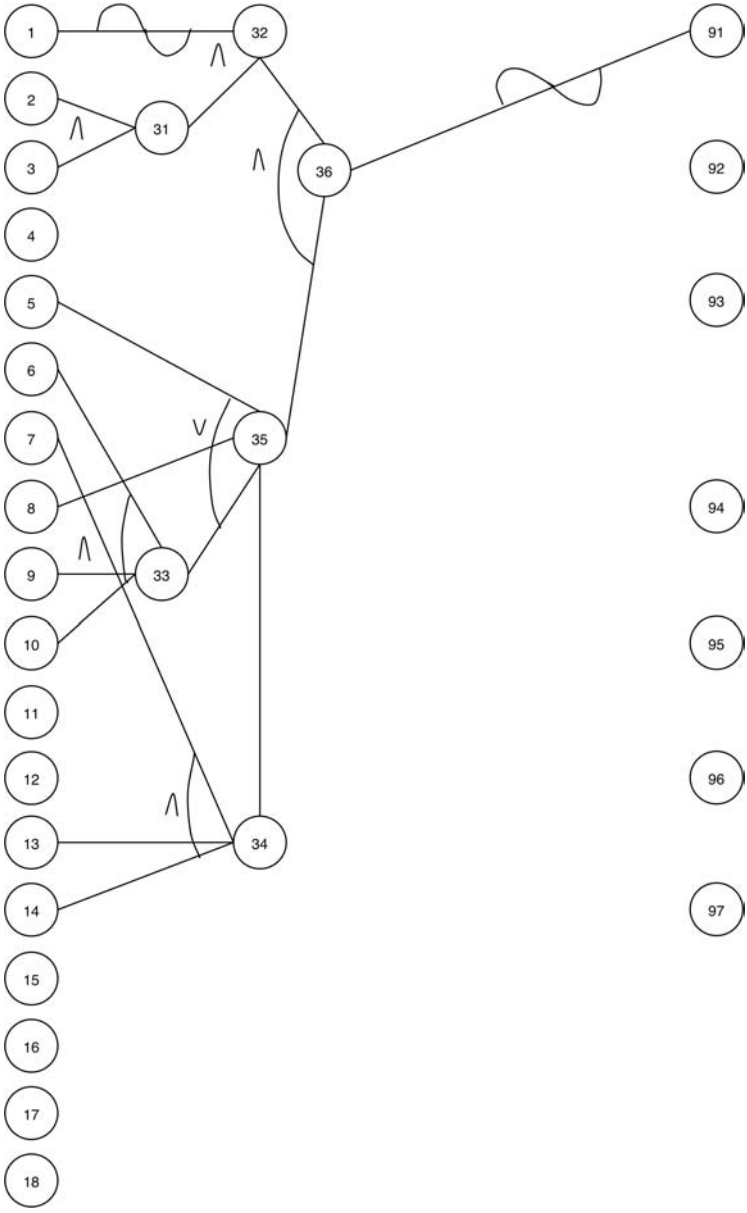


FIGURA 4.12 Início do Gráfico para o Comando DISPLAY.

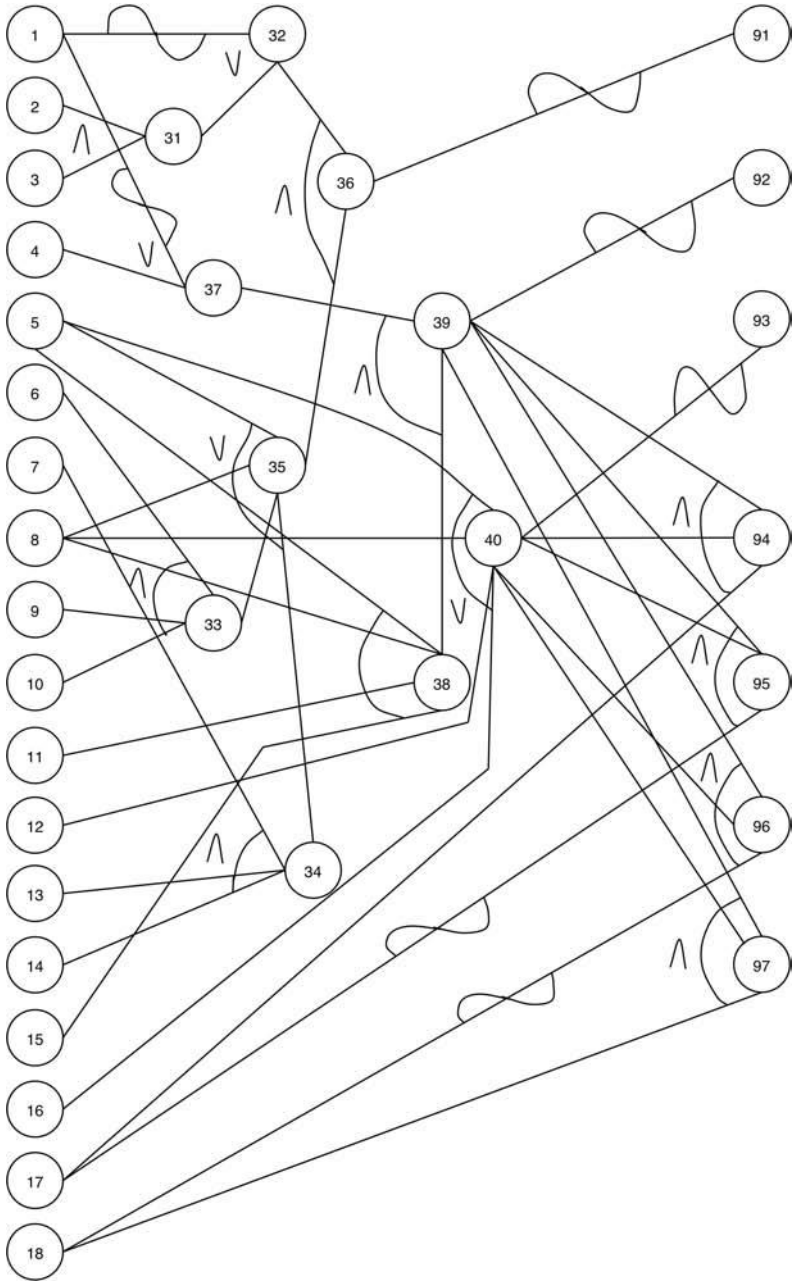


FIGURA 4.13 Gráfico de Causa-Efeito Total sem Constrangimentos.

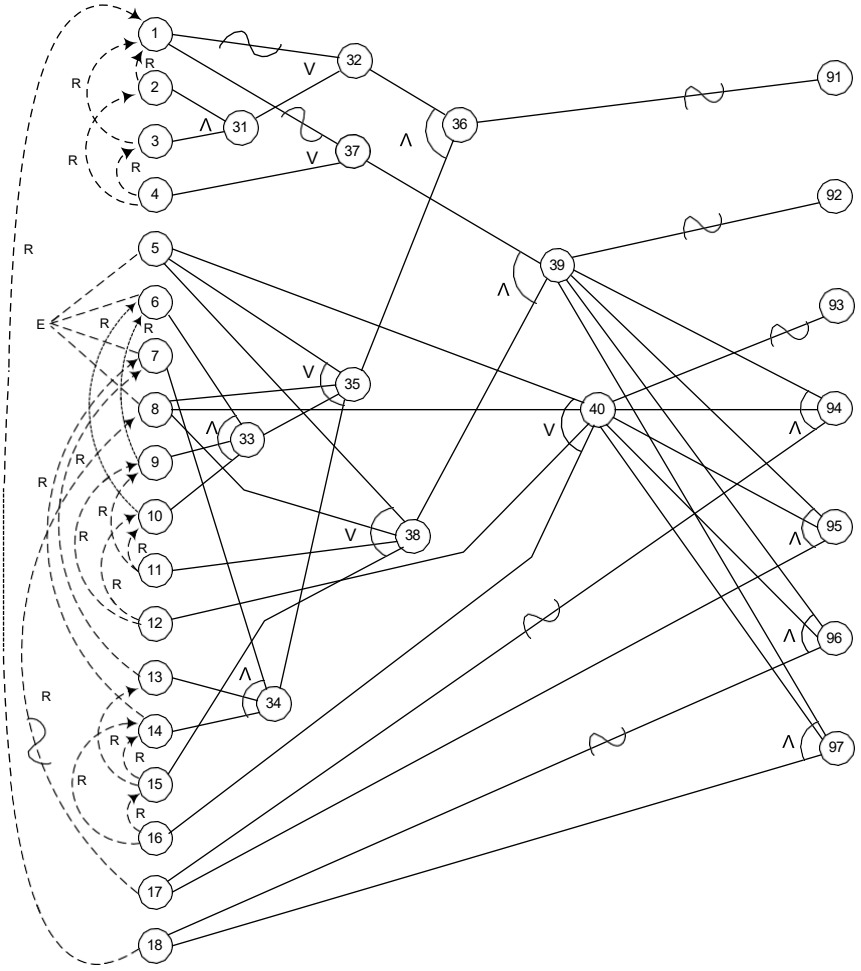


FIGURA 4.14 Gráfico de Causa-Efeito Completo do Comando DISPLAY.

(o segundo operando é omitido); a causa 17 só pode estar presente quando a causa 8 está ausente. Mais uma vez, deve explorar cuidadosamente as condições de constrangimento.

O passo seguinte é a geração de uma tabela de decisão de entrada limitada. Para os leitores familiarizados com as tabelas de decisão, as causas são as condições e os efeitos são as acções. O procedimento utilizado é o seguinte:

1. Seleccionar um efeito para ser o estado presente (1).
2. Rastreando através do gráfico, encontrar todas as combinações de causas (sub-jecto às restrições) que irão definir este efeito para 1.

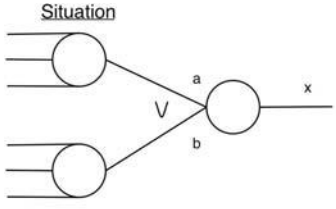
3. Criar uma coluna na tabela de decisão para cada combinação de causas.

4. Para cada combinação, determinar os estados de todos os outros efeitos e colocá-los em cada coluna.

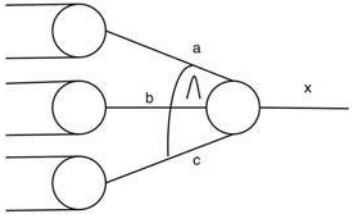
Ao executar a etapa 2, as considerações são as seguintes:

1. Ao rastrear através de um ou nó cuja saída deve ser 1, nunca defina mais do que uma entrada para o ou para 1 simultaneamente. A isto chama-se sensibilizar o caminho. O seu objectivo é evitar a não detecção de certos erros devido a uma causa mascarar outra causa.
2. Ao traçar de volta através de um e nó cuja saída deve ser 0, todas as combinações de entradas que conduzam a 0 saída devem, evidentemente, ser enumeradas. No entanto, se estiver a explorar a situação em que uma entrada é 0 e uma ou mais das outras são 1, não é necessário enumerar todas as condições em que as outras entradas podem ser 1.
3. Ao traçar de volta através de um e nó cuja saída deve ser 0, apenas uma condição em que todas as entradas são zero precisa de ser enumerada. (Se o e está no meio do gráfico de tal forma que as suas entradas vêm de outros nós intermediários, pode haver um número excessivamente grande de situações em que todas as suas entradas são 0).

Estas considerações complicadas estão resumidas na Figura 4.15, e a Figura 4.16 é utilizada como exemplo.



- If x is to be 1, do not bother with the situation where a = b = 1 (consideration one).
- If x is to be 0, enumerate all situations where a = b = 0.



- If x is to be 1, enumerate all situations where a = b = c = 1.
- If x is to be 0, include only one situation where a = b = c = 0 (consideration 3). For the states 001, 010, 100, 011, 101, and 110 of a, b, and c, include only one situation each (consideration 2).

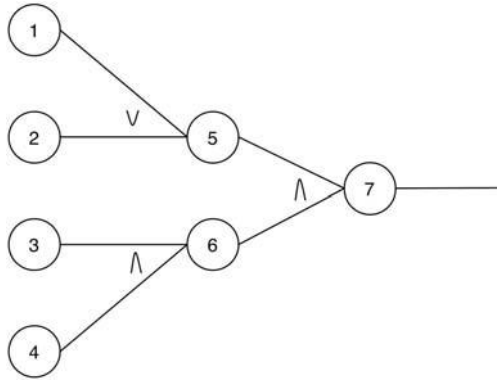


FIGURA 4.16 Gráfico de Amostra para Ilustrar as Considerações de Rastreio.

Assumir que queremos localizar todas as condições de entrada que fazem com que o estado de saída seja 0. A Consideração 3 declara que devemos listar apenas uma circun- posição em que os nós 5 e 6 são 0. A Consideração 2 declara que para o estado em que o nó 5 é 1 e o nó 6 é 0, devemos listar apenas uma circunstância em que o nó 5 é 1, em vez de enumerar todas as formas possíveis em que o nó 5 pode ser 1. Do mesmo modo, para o estado em que o nó 5 é 0 e o nó 6 é 1, devemos listar apenas uma circunstância em que o nó 6 é 1 (embora exista apenas uma neste exemplo). A consideração 1 afirma que, onde o nó 5 deve ser definido como 1, não devemos definir os nós 1 e 2 como 1 simultaneamente. Assim, chegaríamos a cinco estados de nós 1 a 4; por exemplo, os valores:

0	0	0	0	(5¼0, 6¼0)
1	0	0	0	(5¼1, 6¼0)
1	0	0	1	(5¼1, 6¼0)
1	0	1	0	(5¼1, 6¼0)
0	0	1	1	(5¼0, 6¼1)

em vez dos 13 estados possíveis dos nós 1 a 4 que levam a um estado de saída 0.

Estas considerações podem parecer caprichosas, mas têm um propósito im- portante: diminuir os efeitos combinados do gráfico. Eliminam situações que tendem a ser casos de teste de baixo rendimento. Se os casos de teste de baixo rendimento não forem eliminados, um grande gráfico de causa-efeito produzirá um número astronómico de

casos de teste. Se o número de casos de teste for demasiado grande para ser prático,

seleccionará alguns subconjuntos, mas não há garantias de que os casos de teste de baixo rendimento serão os eliminados. Por conseguinte, é melhor eliminá-los durante a análise do gráfico.

Vamos agora converter o gráfico de causa-efeito da Figura 4.14 na tabela de deci- sion. O efeito 91 será seleccionado em primeiro lugar. O Efeito 91 está presente se o nó 36 for
0. O nó 36 é 0 se os nós 32 e 35 forem 0,0; 0,1; ou 1,0; e as considerações 2 e 3 aplicam-se aqui. Rastreando as causas, e considerando as con- straints entre as causas, é possível encontrar as combinações de causas que levam à presença do 91 efeito, embora fazê-lo seja um processo laborioso.

A tabela de decisão resultante, sob a condição de que o efeito 91 esteja presente, é mostrada na Figura 4.17 (colunas 1 a 11). Colunas (testes) de 1 a
3 representam as condições em que o nó 32 é 0 e o nó 35 é 1. Colunas 4 até 10 representam as condições em que o nó 32 é 1 e o nó 35 é 0. Usando a consideração 3, apenas uma situação (coluna 11) de uma possível 21 situações em que os nós 32 e 35 são 0 é identificada. Os espaços em branco na tabela reflectem situações "não importa"(ou seja, o estado da causa é irrelevante) ou indi- cate que o estado de uma causa é óbvio devido aos estados de outras causas dependentes (por exemplo, na coluna 1, sabemos que as causas 5, 7, e 8 devem ser 0 porque existem numa situação "no máximo uma"com causa 6).

As colunas 12 a 15 representam as situações em que o efeito 92 é pres- ent. As colunas 16 e 17 representam as situações em que o efeito 93 está presente. A figura 4.18 representa o resto da tabela de decisão.

O último passo consiste em converter a tabela de decisão em 38 casos de teste. Um conjunto de 38 casos-teste é listado aqui. O número ou números ao lado de cada caso de teste designam os efeitos que se espera que estejam presentes. Assumir que a última localização na memória da máquina a ser utilizada é 7FFF.

1	`VISOR 234AF74-123`	(91)
2	`VISOR 2ZX4-3000`	(91)
3	`EXIBIR HHHHHHHH-2000`	(91)
4	`VISOR 200 200`	(91)
5	`VISOR 0-22222222`	(91)
6	`DISPLAY 1-2X`	(91)
7	`EXIBIR 2-ABCDEFGHI`	(91)
8	`VISOR 3.1111111`	(91)

```
9          DISPLAY 44.$42                (91)
10         MOSTRAR 100.$$$$$$$           (91)
```

	1	2	3	4	5	6	7	8	910	11		12	13	14	15	16	17
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1
4												1	1	0	0	1	1
5				0										1			
6	1	1	1	0	1	1	1				1	1			1	1	
7				0				1	1	1			1				1
8				0													
9	1	1	1		1	0	0				0	1			1	1	
10	1	1	1		0	1	0				1	1			1	1	
11												0			0	1	
12																0	
13								1	0	0			1				1
14								0	1	0			1				1
15													0				
16																	0
17																	
18																	
91	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FIGURA 4.17 Primeira Metade da Tabela de Decisão Resultante.

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1
2	1	1	1	1					1	1	1	1	1	1	1				1	1	1
3	1	1	1	1					1	1	1	1	1	1	1				1	1	1
4	1	1	1	1					1	1	1	1	1	1	1				1	1	1
5	1				1				1				1			1			1		
6			1				1				1			1			1			1	
7				1				1				1			1			1			1
8		1				1				1											
9			1				1				1			1			1			1	
10			1				1				1			1			1			1	
11			1				1				1			1			1			1	
12			1				1				1			1			1			1	
13				1				1				1			1			1			1
14				1				1				1			1			1			1
15				1				1				1			1			1			1
16				1				1				1			1			1			1
17	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
18	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
96	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1
97	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0

FIGURA 4.18 Segunda Metade da Tabela de Decisão Resultante.

11	VISOR 10000000-M	(91)
12	EXIBIR FF-8000	(92)
13	EXIBIR FFF.7001	(92)
14	VISOR 8000-END	(92)
15	VISOR 8000-8001	(92)
16	DISPLAY AA-A9	(93)
17	DISPLAY 7000.0	(93)

19	DISPLAY 1	(94, 97)
20	DISPLAY 21-29	(94, 97)
21	DISPLAY 4021.A	(94, 97)
22	DISPLAY -END	(94, 96)
23	DISPLAY	(94, 96)
24	DISPLAY -F	(94, 96)
25	DISPLAY .E	(94, 96)
26	VISOR 7FF8-END	(94, 96)
27	DISPLAY 6000	(94, 96)
28	DISPLAY A0-A4	(94, 96)
29	DISPLAY 20.8	(94, 96)
30	DISPLAY 7001-END	(95, 97)
31	DISPLAY 5-15	(95, 97)
32w	MOSTRAR 4FF.100	(95, 97)
33	DISPLAY -END	(95, 96)
34	DISPLAY -20	(95, 96)
35	DISPLAY .11	(95, 96)
36	DISPLAY 7000-END	(95, 96)
37	DISPLAY 4-14	(95, 96)
38	DISPLAY 500.11	(95, 96)

Note-se que quando dois ou mais casos de teste diferentes invocaram, na sua maioria, o mesmo conjunto de causas, foram seleccionados valores diferentes para as causas a fim de melhorar ligeiramente o rendimento dos casos de teste. Note-se também que, devido ao tamanho real de armazenamento, o caso de teste 22 é impossível (produzirá efeito 95 em vez de 94, como notado no caso de teste 33). Assim, foram identificados 37 casos de teste.

Observações O gráfico Causa-efeito é um método sistemático de gerar casos de teste representando combinações de condições. A alternativa seria fazer uma selecção ad hoc de combinações; mas, ao fazê-lo, é provável que se ignorassem muitos dos casos de teste 'interessantes'identificados pelo gráfico de causa-efeito.

Uma vez que o gráfico de causa e efeito requer a tradução de uma especificação para uma rede lógica booleana, dá-lhe uma perspectiva

diferente sobre a especificação, e uma visão adicional da mesma. De facto, o desenvolvimento de uma causa...

gráfico de efeito é uma boa forma de descobrir ambiguidades e incompletude nas especificações. Por exemplo, o leitor astuto pode ter notado que este processo descobriu um problema na especificação do comando *DISPLAY*. A especificação afirma que todas as linhas de saída contêm quatro palavras. Isto não pode ser verdade em todos os casos; não pode ocorrer em casos de teste 18 e 26 porque o endereço inicial está a menos de 16 bytes de distância do fim da memória.

Embora o gráfico de causa-efeito produza um conjunto de casos de teste úteis, normalmente não produz todos os casos de teste úteis que possam ser identificados. Por exemplo, no exemplo não dissemos nada sobre a verificação de que os valores da memória mostrados são idênticos aos valores da memória e dissuadir - minar se o programa pode mostrar todos os valores possíveis num local da memória. Além disso, o gráfico de causa-efeito não explora adequadamente as condições de bound- ary. Claro que se poderia tentar cobrir as condições de limite durante o processo. Por exemplo, em vez de identificar a causa única

hexloc2>¼hexloc1

poderia identificar duas causas:

hexloc2 ¼ hexloc1
hexloc2 > hexloc1

O problema em fazer isto, porém, é que complica tremendamente o gráfico e leva a um número excessivamente grande de casos de teste. Por este motivo, é melhor considerar uma análise de valores-limite separada. Por exemplo, as seguintes condições de limite podem ser identificadas para a especificação *DISPLAY:*

1. hexloc1 tem um dígito
2. hexloc1 tem seis dígitos
3. hexloc1 tem sete dígitos
4. hexloc1 ¼ 0
5. hexloc1 ¼ 7FFF
6. hexloc1 ¼ 8000
7. hexloc2 tem um dígito
8. hexloc2 tem seis dígitos
9. hexloc2 tem sete dígitos
10. hexloc2 ¼ 0

11. hexloc2 ¼ 7FFF
12. hexloc2 ¼ 8000
13. hexloc2 ¼ hexloc
14. hexloc2 ¼ hexloc1 þ 1
15. hexloc2 ¼ hexloc1 − 1
16. bytecount tem um dígito
17. bytecount tem seis dígitos
18. bytecount tem sete dígitos
19. bytecount ¼ 1
20. hexloc1 þ bytecount ¼ 8000
21. hexloc1 þ bytecount ¼ 8001
22. exibir 16 bytes (uma linha)
23. exibir 17 bytes (duas linhas)

Note-se que isto não implica que escreva 60 (37 þ 23) casos de teste. Uma vez que o gráfico de causa-efeito nos dá margem de manobra na selecção de val- ues específicos para operandos, as condições limite poderiam ser misturadas nos casos de teste derivados do gráfico de causa-efeito. Neste exemplo, reescrevendo alguns dos 37 casos de teste originais, todas as 23 condições limite poderiam ser cobertas sem quaisquer casos de teste adicionais. Assim, chegamos a um conjunto pequeno mas potente de casos de teste que satisfazem ambos os objectivos.

Note-se que o gráfico de causa-efeito é consistente com vários dos princípios de teste do Capítulo 2. A identificação do resultado esperado de cada caso de teste é uma parte inerente da técnica (cada coluna na tabela de decisão indi- cata os efeitos esperados). Note-se também que nos encoraja a procurar efeitos secundários não desejados. Por exemplo, a coluna (teste) 1 especifica que devemos esperar a presença do efeito 91 e que os efeitos 92 a 97 devem estar ausentes.

O aspecto mais difícil da técnica é a conversão do gráfico na tabela de decisão. Este processo é algorítmico, implicando que se poderia automatizá-lo escrevendo um programa; existem vários programas comerciais para ajudar com a conversão.

Adivinhação de erros

Tem-se notado frequentemente que algumas pessoas parecem ser naturalmente hábeis em testes pro-grama. Sem utilizar qualquer metodologia específica, como por exemplo, o limite

análise de valor do gráfico de causa-efeito, estas pessoas parecem ter um dom para farejar os erros.

Uma explicação para isto é que estas pessoas estão a praticar-sub-conscientemente mais frequentemente do que não - uma técnica de concepção de casos de teste que poderia ser chamada de adivinhação de erros. Dado um programa específico, elas supõem - tanto por intuição e experiência - certos tipos prováveis de erros e depois escrevem casos de teste para exporem esses erros.

É difícil dar um procedimento para a técnica de adivinhação de erros, uma vez que é em grande parte um processo intuitivo e ad hoc. A ideia básica é enumerar uma lista de possíveis erros ou situações propensas a erros e depois escrever casos de teste com base na lista. Por exemplo, a presença do valor 0 numa entrada pro-grama é uma situação propícia a erros. Portanto, pode escrever casos de teste para os quais determinados valores de entrada têm um valor 0 e para os quais os valores de saída par- ticular são forçados a 0. Também, onde um número variável de entradas ou saídas pode estar presente (por exemplo, o número de entradas numa lista a ser pesquisada), os casos de "nenhum" e "um" (por exemplo, lista vazia, lista contendo apenas uma entrada) são situações propensas a erros. Outra ideia é identificar casos de teste associados a suposições que o programador possa ter feito ao ler a especificação (ou seja, factores que foram omitidos da especificação, seja por acidente ou porque o escritor os sentiu como óbvios).

Uma vez que um procedimento de adivinhação de erros não pode ser dado, o próximo melhor al- ternative é discutir o espírito da prática, e a melhor maneira de o fazer é apresentando exemplos. Se se estiver a testar uma sub-rotina de triagem, a dobra-abaixamento são situações a explorar:

- A lista de entrada está vazia.
- A lista de entrada contém uma entrada.
- Todas as entradas da lista de entrada têm o mesmo valor.
- A lista de entrada já está ordenada.

Por outras palavras, enumera os casos especiais que podem ter sido ignorados quando o programa foi concebido. Se estiver a testar uma sub-rotina de pesquisa binária, poderá tentar as situações em que: (1) existe apenas uma entrada na tabela a ser pesquisada; (2) o tamanho da tabela é uma potência de 2 (por exemplo, 16); e (3) o tamanho da tabela é uma potência inferior e uma potência superior a 2 (por exemplo, 15 ou 17).

Considerar o programa MTEST na secção sobre análise de valores-limite. Os seguintes testes adicionais vêm à mente quando se utiliza a técnica de adivinhação de erros:

- O programa aceita "em branco" como resposta?
- Um registo de tipo 2 (resposta) aparece no conjunto de tipo 3 (estudante) registos.
- Um registo sem um 2 ou 3 na última coluna aparece como diferente do registo inicial (título).
- Dois estudantes têm o mesmo nome ou número.
- Uma vez que uma mediana é calculada de forma diferente dependendo da existência ou não de
  um número ímpar ou par de itens, testar o programa para um número par de estudantes e um número ímpar de estudantes.
- O campo do número de perguntas tem um valor negativo.

Os testes de adivinhação de erros que me vêm à mente para o comando `DISPLAY` da secção anterior são os seguintes:

`DISPLAY 100-` (segundo operando parcial)
`DISPLAY 100.` (segundo operando parcial)
`DISPLAY 100-10A 42` (segundo operando extra) `DISPLAY 000-0000FF` (zeros à cabeça)

A Estratégia

As metodologias de concepção de casos de teste discutidas neste capítulo podem ser com- cingidas a uma estratégia global. A razão para as combinar já deveria ser óbvia: Cada uma contribui com um conjunto particular de casos de teste úteis, mas nenhuma delas por si só contribui com um conjunto completo de casos de teste. Uma estratégia razoável é a seguinte:

1. Se a especificação contiver combinações de condições de entrada, comece com o gráfico de causa-efeito.
2. Em qualquer caso, utilizar a análise do valor limite. Lembre-se que esta é uma análise dos limites de entrada e saída. A análise do valor-limite produz um conjunto de condições de teste suplementares, mas, como se observa no gráfico de segundo

parágrafo sobre causa-efeito, muitos ou todos estes podem ser classificados incorpóreos nos testes de causa-efeito.

3. Identificar as classes de equivalência válidas e inválidas para a entrada e saída, e complementar os casos de teste identificados acima, se necessário.
4. Utilizar a técnica de adivinhação de erros para acrescentar casos de teste adicionais.
5. Examinar a lógica do programa no que respeita ao conjunto de casos de teste. Utilizar o critério de cobertura de decisão, cobertura de condição, cobertura de decisão/condição - idade, ou cobertura de condições múltiplas (sendo o último o mais completo). Se o critério de cobertura não tiver sido cumprido pelos casos de teste identificados nas quatro etapas anteriores, e se o cumprimento do critério não for impossível (ou seja, certas combinações de condições podem ser im- possíveis de criar devido à natureza do programa), adicionar casos de teste suficientemente cientados para fazer com que o critério seja cumprido.

Mais uma vez, a utilização desta estratégia não garantirá que todos os erros serão encontrados, mas foi considerado como representando um compromisso razoável. Além disso, representa uma quantidade considerável de trabalho árduo, mas como dissemos no início deste capítulo, nunca ninguém afirmou que os testes do programa são fáceis.

Resumo

Uma vez acordado que os testes de software agressivos são um complemento digno dos seus esforços de desenvolvimento, o passo seguinte é conceber casos de teste que irão exercer a sua aplicação o suficiente para produzir resultados de teste satisfatórios. Na maior parte dos casos, considere uma combinação de methodol- ogies de caixa negra e de caixa branca para garantir que concebeu testes de programa rigorosos.

As técnicas de concepção de casos de teste discutidas neste capítulo incluem:

- Cobertura lógica. Testes que exercem todos os resultados dos pontos de decisão pelo menos uma vez, e asseguram que todas as declarações ou pontos de entrada são executados pelo menos uma vez.
- Particionamento por equivalência. Define classes de condição ou erro para ajudar a re

duce o número de testes finitos. Presume-se que um teste de um valor representativo dentro de uma classe também testa todos os valores ou

condições dentro dessa classe.

- Análise do valor-limite. Testa cada condição de borda de uma equivalência classe; considera também as classes de equivalência de saída, bem como as classes de entrada.
- Gráfico de causa-efeito. Produz representações gráficas booleanas de potenciais resultados de casos de teste para ajudar na selecção de casos de teste eficientes e completos.

- Adivinhação de erros. Produz casos de teste baseados no conhecimento intuitivo e especializado dos membros da equipa de teste para definir potenciais erros de software para facilitar a concepção eficiente de casos de teste.

Testes extensos e aprofundados não são fáceis; nem o desenho de casos de teste mais extenso assegurará que todos os erros serão descobertos. Dito isto, os programadores dispostos a ir além dos testes rápidos, que dedicarão tempo suficiente para testar a concepção de casos, analisar cuidadosamente os resultados dos testes, e agir de forma decisiva sobre os resultados, serão recompensados com software funcional, fiável e livre de erros.