4 Projeto de Caso de Teste

Indo além das questões psicológicas discutidas no Capítulo 2, a consideração mais importante no teste de programa é o design e a criação de casos de teste eficazes.

Testes, por mais criativos e aparentemente completos, não podem garantir a ausência de todos os erros. O design do caso de teste é tão importante porque o teste completo é impossível. Dito de outra forma, um teste de qualquer programa deve ser necessariamente incompleto. A estratégia óbvia, então, é tentar fazer os testes o mais completos possível.

Dadas as restrições de tempo e custo, a principal questão do teste se torna:

Qual subconjunto de todos os casos de teste possíveis tem a maior probabilidade de detectar a maioria dos erros?

O estudo de metodologias de projeto de casos de teste fornece respostas para essa pergunta.

Em geral, a metodologia menos eficaz de todas é o teste de entrada aleatória – o processo de testar um programa selecionando, aleatoriamente, algum subconjunto de todos os valores de entrada possíveis. Em termos de probabilidade de detectar a maioria dos erros, uma coleção de casos de teste selecionados aleatoriamente tem pouca chance de ser um subconjunto ótimo, ou mesmo próximo ao ótimo. Portanto, neste capítulo, queremos desenvolver um conjunto de processos de pensamento que permitem selecionar dados de teste de forma mais inteligente.

O Capítulo 2 mostrou que testes exaustivos de caixa preta e caixa branca são, em geral, impossível; ao mesmo tempo, sugeriu que uma estratégia de teste razoável poderia apresentar elementos de ambos. Esta é a estratégia desenvolvida neste capítulo. Você pode desenvolver um teste razoavelmente rigoroso usando certas metodologias de projeto de caso de teste orientadas a caixa preta e, em seguida, complementando esses casos de teste examinando a lógica do programa, usando caixa branca métodos.

As metodologias discutidas neste capítulo são:

Caixa preta	Caixa branca
Particionamento equivalente	Cobertura do extrato
Análise de valor de limite	Cobertura da decisão
Gráficos de causa e efeito	Cobertura de condição
Erro ao adivinhar	Cobertura de decisão/condição
	Cobertura de várias condições

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

Ninguém jamais prometeu que o teste de software seria fácil. Para citar um velho sábio, "Se você achava que projetar e codificar esse programa era difícil, você ainda não vi nada."

O procedimento recomendado é desenvolver casos de teste usando os métodos da caixa preta e, em seguida, desenvolver casos de teste complementares, conforme necessário, com métodos de caixa branca. Discutiremos a caixa branca mais conhecida métodos primeiro.

Teste de caixa branca

O teste de caixa branca está preocupado com o grau em que os casos de teste exercem ou cobrem a lógica (código-fonte) do programa. Como vimos no Capítulo 2, o teste final da caixa branca é a execução de cada caminho no programa; mas o teste de caminho completo não é uma meta realista para um programa com laços.

Teste de cobertura lógica

Se você se afastar completamente do teste de caminho, pode parecer que uma meta válida seria executar todas as instruções do programa pelo menos uma vez. Infelizmente, este é um critério fraco para um teste de caixa branca razoável. Este conceito é ilustrado na Figura 4.1. Suponha que esta figura represente um pequeno programa a ser testado. O trecho de código Java equivalente segue:

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

    } if(A¼¼2 jj X>1) {
        X¼X+1;
    }
}
```

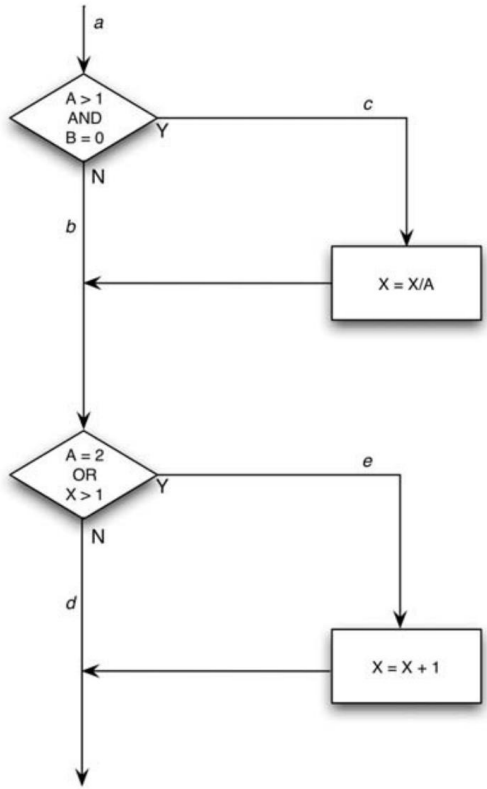

FIGURA 4.1 Um pequeno programa a ser testado.

Você pode executar cada instrução escrevendo um único caso de teste que percorre o caminho ace. Ou seja, definindo A¼2, B¼0 e X¼3 no ponto a, cada instrução seria executada uma vez (na verdade, X poderia receber qualquer valor inteiro >1).

Infelizmente, este critério é bastante pobre. Por exemplo, talvez a primeira decisão deva ser um ou em vez de um e. Nesse caso, esse erro não seria detectado. Talvez a segunda decisão devesse ter declarado X>0; este erro não seria detectado. Além disso, há um caminho pelo programa no qual X permanece inalterado (o caminho abd). Se isso fosse um erro, passaria despercebido. Em 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 ramificação. Esse critério afirma que você deve escrever casos de teste suficientes para que cada decisão tenha um resultado verdadeiro e um resultado falso pelo menos uma vez. Em outras palavras, cada direção de ramificação deve ser percorrida pelo menos uma vez. Exemplos de instruções de ramificação ou decisão são instruções switch-case, do-while e if-else . As instruções GOTO de vários caminhos se qualificam em algumas linguagens de programação, como Fortran.

A cobertura de decisão geralmente pode satisfazer a cobertura de instrução. Como cada instrução está em algum subcaminho que emana de uma instrução de ramificação ou do ponto de entrada do programa, cada instrução deve ser executada se todas as direções de ramificação forem executadas. Existem, no entanto, pelo menos três exceções:

Programas sem decisões.
Programas ou sub-rotinas/métodos com múltiplos pontos de entrada. Uma determinada instrução pode ser executada apenas se o programa for inserido em um ponto de entrada específico.
Declarações dentro das unidades ON. Atravessar todas as direções de ramificação não necessariamente fará com que todas as unidades ON sejam executadas.

Como consideramos a cobertura de declaração uma condição necessária, a cobertura de decisão, um critério aparentemente melhor, deve ser definida para incluir a cobertura de declaração. Assim, a cobertura de decisão requer que cada decisão tenha um resultado verdadeiro e um falso, e que cada afirmação seja executada pelo menos uma vez. Uma maneira alternativa e mais fácil de expressar isso é que cada decisão tem um resultado verdadeiro e um falso, e que cada ponto de entrada (incluindo unidades ON) seja invocado pelo menos uma vez.

Essa discussão considera apenas decisões ou ramificações bidirecionais e deve ser modificada para programas que contêm decisões de vários caminhos. Exemplos são programas Java contendo instruções switch-case , programas Fortran contendo instruções IF aritméticas (três vias) ou instruções GOTO computadas ou aritméticas e programas COBOL contendo instruções GOTO alteradas ou instruções GO-TO-DEPENDING-ON . Para tais programas, o critério é exercitar cada resultado possível de todas as decisões pelo menos uma vez e invocar cada ponto de entrada do programa ou sub-rotina pelo menos uma vez.

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

A cobertura de decisão é um critério mais forte do que a cobertura de declaração, mas ainda é bastante fraca. Por exemplo, há apenas 50% de chance de explorarmos o caminho onde x não é alterado (ou seja, somente se escolhermos a primeira alternativa). Se a segunda decisão estivesse errada (se deveria ter dito X<1 em vez de X>1), o erro não seria detectado pelos dois casos de teste no exemplo anterior.

Um critério que às vezes é mais forte do que a cobertura de decisão é a cobertura de condição. Nesse caso, você escreve casos de teste suficientes para garantir que cada condição em uma decisão assuma todos os resultados possíveis pelo menos uma vez. Mas, como na cobertura de decisão, isso nem sempre leva à execução de cada instrução, portanto, uma adição ao critério é que cada ponto de entrada do programa ou sub-rotina, bem como as unidades ON, sejam invocados pelo menos uma vez. Por exemplo, a declaração de ramificação:

FAÇA K¼0 a 50 ENQUANTO (JþK<QUEST)

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

A Figura 4.1 tem quatro condições: A>1, B¼0, A¼2 e X>1. Portanto, casos de teste suficientes são necessários 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 são presente no ponto b. Um número suficiente de casos de teste que satisfaçam o critério e os caminhos percorridos por cada um são:

A¼2, B¼0, X¼4 ás
A¼1, B¼1, X¼1 adb

Observe que, embora o mesmo número de casos de teste tenha sido gerado para este exemplo, a cobertura de condição geralmente é superior à cobertura de decisão, pois pode (mas nem sempre) fazer com que cada condição individual em uma decisão seja executada com ambos os resultados, enquanto a cobertura de decisão não. Por exemplo, na mesma declaração de ramificação

FAÇA K¼0 a 50 ENQUANTO (JþK<QUEST)

é uma ramificação de duas vias (execute o corpo do loop ou pule-o). Se você estiver usando o teste de decisão, o critério pode ser satisfeito deixando o loop correr de K¼0 a 51, sem nunca explorar a circunstância em que a cláusula WHILE se torna falsa. Com o critério de condição, entretanto, um caso de teste seria necessário para gerar um resultado falso para as condições JþK<QUEST.

Embora o critério de cobertura de condição pareça, à primeira vista, satisfazer o critério de cobertura de decisão, nem sempre o faz. Se a decisão SE(A & B) estiver sendo testada, o critério de cobertura de condição permitiria escrever dois casos de teste – A é verdadeiro, B é falso e A é falso, B é verdadeiro – mas isso não causaria o ENTÃO cláusula do IF a ser executado. Os testes de cobertura de condição para o exemplo anterior cobriram todos os resultados da decisão, mas isso foi apenas por acaso. Por exemplo, dois casos de teste alternativos

A¼1, B¼0, X¼3
A¼2, B¼1, X¼1

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

A saída óbvia para esse dilema é um critério chamado cobertura de decisão/condição. Requer casos de teste suficientes para que cada condição em uma decisão receba todos os resultados possíveis pelo menos uma vez, cada decisão receba todos os resultados possíveis pelo menos uma vez e cada ponto de entrada seja invocado pelo menos uma vez.

Um ponto fraco da cobertura de decisão/condição é que, embora possa parecer exercer todos os resultados de todas as condições, frequentemente não o faz, porque certas condições mascaram outras condições. Para ver isso, examine a Figura 4.2. O fluxograma nesta figura é a maneira como um compilador geraria código de máquina para o programa da Figura 4.1. As decisões multicondicionais no programa fonte foram divididas em decisões e ramificações individuais porque a maioria das máquinas não possui uma única instrução que toma decisões multicondicionais. Uma cobertura de teste mais completa, então,

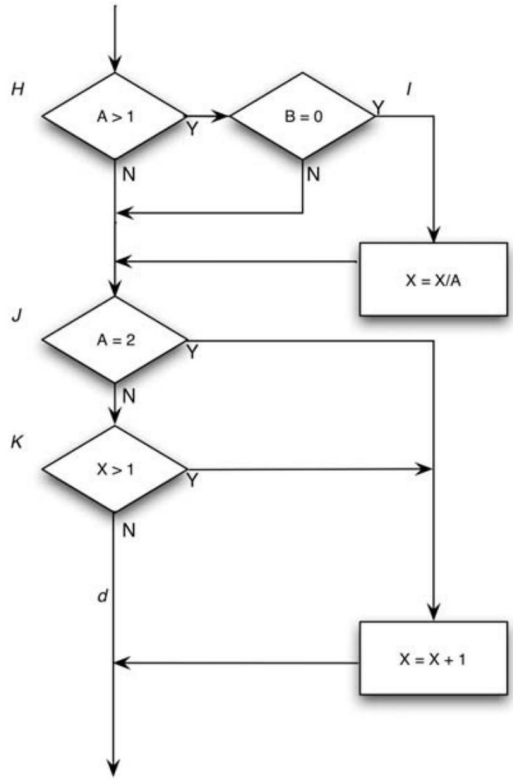

FIGURA 4.2 Código de Máquina para o Programa da Figura 4.1.

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

A razão, como mostrado na Figura 4.2, é que os resultados das condições no e as expressões ou podem mascarar ou bloquear a avaliação de outras condições. Por exemplo, se uma condição e for falsa, nenhuma das
condições na expressão precisam ser avaliadas. Da mesma forma, se uma condição ou for verdadeira, nenhuma das condições subsequentes precisa ser avaliada. Por isso, erros em expressões lógicas não são necessariamente revelados pela condição critérios de cobertura e de cobertura de decisão/condição.

Um critério que cobre este problema, e mais alguns, é a condição múltipla cobertura. Este critério requer que você escreva casos de teste suficientes para que todas as combinações possíveis de resultados de condição em cada decisão, e todas

pontos de entrada, são invocados pelo menos uma vez. Por exemplo, considere a seguinte sequência de pseudocódigo.

```
NOTFOUND¼TRUE;
DO I¼1 para TABSIZE WHILE (NOTFOUND); /*PESQUISAR TABELA*/
    .. . procurando lógica .. .          ;
FIM
```

As quatro situações a serem testadas são:

1. I<¼TABSIZE e NOTFOUND são verdadeiros.
2. I<¼TABSIZE e NOTFOUND é falso (encontrar a entrada antes bater na ponta da mesa).
3. I>TABSIZE e NOTFOUND são verdadeiros (atingir o final da tabela sem encontrar a entrada).
4. I>TABSIZE e NOTFOUND é falso (a entrada é a última no tabela).

Deve ser fácil ver que um conjunto de casos de teste que satisfaça o critério de condição múltipla também satisfaz a cobertura de decisão, cobertura de condição, e critérios de cobertura de decisão/condição.

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

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

Nota Lembre-se do trecho de código Java apresentado anteriormente que os casos de teste 5 a 8 expressam valores no ponto da segunda instrução if . Desde X pode ser alterado acima desta instrução if , os valores necessários nesta instrução if devem ser copiados através da lógica para encontrar a entrada correspondente valores.

Essas combinações a serem testadas não implicam necessariamente que oito casos são necessários. Na verdade, eles podem ser cobertos por quatro casos de teste. Os valores de entrada do caso de teste e as combinações que eles cobrem são os seguintes:

A¼2, B¼0, X¼4 Coberturas 1, 5

A¼2, B¼1, X¼1 Coberturas 2, 6

A¼1, B¼0, X¼2 Coberturas 3, 7
A¼1, B¼1, X¼1 Coberturas 4, 8

O fato de haver quatro casos de teste e quatro caminhos distintos na Figura 4.1 é apenas coincidência. Na verdade, esses quatro casos de teste não cobrem todos os caminhos; eles perdem o caminho acd. Por exemplo, você precisaria de oito casos de teste para a seguinte decisão:

```
if(x¼¼y && comprimento(z)¼¼0 && FLAG)
        { j¼1;
senão
        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 menor que o número de caminhos.

Em resumo, para programas contendo apenas uma condição por decisão, um critério mínimo de teste 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 ON) pelo menos uma vez, para garantir que todas as instruções sejam executadas pelo menos uma vez. Para programas contendo decisões com múltiplas condições, o critério mínimo é um número suficiente de casos de teste para invocar todas as combinações possíveis de resultados de condição 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 caixa preta

Conforme discutimos no Capítulo 2, o teste de caixa preta (orientado por dados ou orientado por entrada/saída) é baseado nas especificações do programa. O objetivo é encontrar áreas em que o programa não se comporte de acordo com suas especificações.

Particionamento equivalente

O Capítulo 2 descreveu um bom caso de teste como aquele que tem uma probabilidade razoável de encontrar um erro; também afirmou que um teste exaustivo de entrada de um programa é impossível. Portanto, ao testar um programa, você está limitado a um

pequeno subconjunto de todas as entradas possíveis. Claro, então, você deseja selecionar o subconjunto "certo", ou seja, o subconjunto com a maior probabilidade de encontrar o maioria dos erros.

Uma maneira de localizar esse subconjunto é perceber que um caso de teste bem selecionado também deve ter duas outras propriedades:

1. Reduz, em mais de um, o número de outros casos de teste que devem ser desenvolvidos para atingir algum objetivo predefinido de teste "razoável".

2. Abrange um grande conjunto de outros casos de teste possíveis. Ou seja, ele nos diz algo sobre a presença ou ausência de erros além desse conjunto específico de valores de entrada.

Essas propriedades, embora pareçam semelhantes, descrevem duas considerações distintas. A primeira implica que cada caso de teste deve invocar tantas considerações de entrada diferentes quanto possível para minimizar o número total de casos de teste necessários. A segunda implica que você deve tentar particionar o domínio de entrada de um programa em um número finito de classes de equivalência tal que você possa razoavelmente supor (mas, é claro, não ter certeza absoluta) que um teste de um valor representativo de cada class é equivalente a um teste de qualquer outro valor. Ou seja, se um caso de teste em uma classe de equivalência detecta um erro, espera-se que todos os outros casos de teste na classe de equivalência encontrem o mesmo erro. Por outro lado, se um caso de teste não detectou um erro, esperaríamos que nenhum outro caso de teste na classe de equivalência se enquadrasse em outra classe de equivalência, uma vez que as classes de equivalência podem se sobrepor.

Essas duas considerações formam uma metodologia de caixa preta conhecida como particionamento de equivalência. A segunda consideração é usada para desenvolver um conjunto de condições “interessantes” a serem testadas. A primeira consideração é então usada para desenvolver um conjunto mínimo de casos de teste cobrindo essas condições.

Um exemplo de uma classe de equivalência no programa de triângulos do Capítulo 1 é o conjunto "três números de igual valor com valores inteiros maiores que zero". um teste de um elemento do conjunto, é improvável que um erro seja encontrado por um teste de outro elemento do conjunto. Em outras palavras, nosso tempo de teste é melhor gasto em outro lugar: em diferentes classes de equivalência.

O design do caso de teste por particionamento de equivalência prossegue em duas etapas: (1) identificar as classes de equivalência e (2) definir os casos de teste.

Condição externa	Classes de equivalência válidas	Classes de equivalência inválidas

FIGURA 4.3 Um Formulário para Enumerar Classes de Equivalência.

Identificando as Classes de Equivalência As classes de equivalência são identificadas tomando cada condição de entrada (geralmente uma sentença ou frase na especificação) e dividindo-a em dois ou mais grupos. Você pode usar a tabela na Figura 4.3 para fazer isso. Observe que dois tipos de classes de equivalência são identificados: classes de equivalência válidas representam entradas válidas para o programa e classes de equivalência inválidas representam todos os outros estados possíveis da condição (ou seja, valores de entrada errôneos). Assim, estamos aderindo ao princípio 5, discutido no Capítulo 2, que afirma que você deve focar a atenção em condições inválidas ou inesperadas.

Dada uma entrada ou condição externa, identificar as classes de equivalência é em grande parte um processo heurístico. Siga estas orientações:

1. Se uma condição de entrada especificar um intervalo de valores (por exemplo, "a contagem de itens pode ser de 1 a 999"), identifique 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, "de 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 motivos para acreditar que o programa trata cada um de forma diferente ("tipo

de veículo deve ser ÔNIBUS, CAMINHÃO, TÁXICA, PASSAGEIRO ou MOTO"), identificar uma classe de equivalência válida para cada uma e uma classe de equivalência inválida ("TRAILER", por exemplo).

4. Se uma condição de entrada especificar uma situação "obrigatória", como "o primeiro caractere do identificador deve ser uma letra", identifique uma classe de equivalência válida (é uma letra) e uma classe de equivalência inválida (não é uma carta).

Se houver alguma razão para acreditar que o programa não trata os elementos de uma classe de equivalência de forma idêntica, divida a classe de equivalência em classes de equivalência menores. Ilustraremos um exemplo desse processo em breve.

Identificando os Casos de Teste O segundo passo é o uso de classes de equivalência para identificar os casos de teste. O processo é como se segue:

1. Atribua um número único a cada classe de equivalência.
2. Até que todas as classes de equivalência válidas tenham sido cobertas (incorporadas em) casos de teste, escreva um novo caso de teste cobrindo o maior número possível de classes de equivalência válidas descobertas.
3. Até que 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 descobertas.

A razão pela qual os casos de teste individuais cobrem casos inválidos é que certos verificações de entrada errônea mascaram ou substituem outras verificações de entrada errônea. Por exemplo, se a especificação indicar "digite o tipo de livro (HARDCOVER, SOFTCOVER ou LOOSE) e o valor (1-999)," o caso de teste, (XYZ 0), expressando duas condições de erro (tipo e valor do livro inválidos) provavelmente não exercerá o cheque do valor, pois o programa pode dizer
"XYZ É TIPO DE LIVRO DESCONHECIDO" e não se preocupe em examinar os restos da entrada.

Um exemplo

Como exemplo, suponha que estamos desenvolvendo um compilador para um subconjunto da linguagem Fortran e desejamos testar a verificação de sintaxe da instrução DIMENSION . A especificação está listada abaixo. (Nota: isso não é

a declaração completa do Fortran DIMENSION ; foi editado consideravelmente para torná-lo tamanho de livro didático. Não se iluda pensando que o teste de programas reais é tão fácil quanto os exemplos deste livro.) Na especificação, os itens em itálico indicam unidades sintáticas para as quais entidades específicas devem ser substituídas em declarações reais; colchetes são usados para indicar itens de opção; e uma reticência indica que o item anterior pode aparecer várias vezes em sucessão.

Uma instrução DIMENSION é usada para especificar as dimensões das matrizes. A forma da instrução DIMENSION é

DIMENSION anúncio[,anúncio]...

onde ad é um descritor de matriz do formulário

n(d[,d]...)

onde n é o nome simbólico da matriz e d é um declarador de dimensão. Os nomes simbólicos podem ter de uma a seis letras ou dígitos, sendo que o primeiro deve ser uma letra. Os números mínimo e máximo de 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 de dimensão inferior e superior. Um limite pode ser uma constante no intervalo de 65534 a 65535 ou o nome de uma variável inteira (mas não um nome de 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, seu valor pode ser negativo, 0 ou positivo. Como para todas as instruções, a instrução DIMENSION pode ser continuada em várias linhas.

O primeiro passo é identificar as condições de entrada e, a partir delas, localizar as classes de equivalência. Estes são tabulados na Tabela 4.1. Os números na tabela são identificadores únicos das classes de equivalência.

O próximo passo é escrever um caso de teste cobrindo uma ou mais classes de equivalência válidas. Por exemplo, o caso de teste

DIMENSÃO A(2)

abrange 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 matriz descritores	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 algo mais (9)
O nome do array começa com a letra	sim (10)	não (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), outra coisa (18)
Nome da variável inteira	tem a letra (19), tem dígitos (20)	tem outra coisa (21)
A variável inteira começa com carta	sim (22)	não (23)
Constante	–65534–65535 (24)	<–65534 (25), >65535 (26)
Limite inferior especificado	sim (27), não (28)	
Limite superior para inferior vinculado	maior que (29), igual (30)	menor que (31)
Limite inferior especificado	negativo (32), zero (33), > 0 (34)	
O limite inferior é	constante (35), variável inteira (36)	nome do elemento da matriz (37), outra coisa (38)
O limite inferior é	um (39)	ub>¼1 (40), ub<1 (41)
Várias linhas	sim (42), não (43)	

O próximo passo é elaborar um ou mais casos de teste cobrindo o restante 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)
```

abrange as demais classes. As classes de equivalência de entrada inválidas e um caso de teste representando cada um, 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)
(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 cobertas por 17 casos de teste. Você pode querer considerar como esses casos de teste se compararam a um conjunto de casos de teste derivados de maneira ad hoc.

Embora o particionamento de equivalência seja muito superior a uma seleção aleatória de casos de teste, ele ainda apresenta deficiências. Ele ignora certos tipos de casos de teste de alto rendimento, por exemplo. As próximas duas metodologias, análise de valor limite e gráficos de causa e efeito, cobrem muitas dessas deficiências.

Análise de valor de limite

A experiência mostra que os casos de teste que exploram as condições de contorno têm um retorno maior do que os casos de teste que não exploram. As condições de fronteira são aquelas situações diretamente sobre, acima e abaixo das bordas das classes de equivalência de entrada e classes de equivalência de saída. A análise de valor limite difere da partição de equivalência em dois aspectos:

1. Em vez de selecionar qualquer elemento em uma classe de equivalência como sendo representativo, a análise de valor de contorno requer que um ou mais elementos sejam selecionados de modo que cada aresta da classe de equivalência seja objeto de um teste.
2. Em vez de apenas focar a atenção nas condições de entrada (espaço de entrada), os casos de teste também são derivados considerando o espaço de resultado (classes de equivalência de saída).

É difícil apresentar um "livro de receitas" para a análise de valor de fronteira, pois requer um grau de criatividade e uma certa especialização em relação ao problema em questão. (Portanto, como muitos outros aspectos do teste, é mais um estado de espírito do que qualquer outra coisa.) No entanto, algumas diretrizes gerais são necessárias:

1. Se uma condição de entrada especificar um intervalo de valores, escreva casos de teste para as extremidades do intervalo e casos de teste de entrada inválida para situações logo além das extremidades. Por exemplo, se o domínio válido de um valor de entrada for –1,0 a 1,0, escreva 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, escreva casos de teste para o número mínimo e máximo de valores e um abaixo e além desses valores. Por exemplo, se um arquivo de entrada pode conter de 1 a 255 registros, escreva casos de teste para 0, 1, 255 e 256 registros.
3. Use a diretriz 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, escreva casos de teste que causem a dedução de $ 0,00 e $ 1.165,25. Além disso, veja se é possível inventar casos de teste que possam causar uma dedução negativa ou uma dedução de mais de $ 1.165,25.

   Observe que é importante examinar os limites do espaço de resultados porque nem sempre os limites dos domínios de entrada representam o mesmo conjunto de circunstâncias que os limites dos intervalos 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. Use a diretriz 2 para cada condição de saída. Se um sistema de recuperação de informações exibe os resumos mais relevantes com base em uma solicitação de entrada, mas nunca mais de quatro resumos, escreva casos de teste de modo que o programa exiba zero, um e quatro resumos e escreva um caso de teste que possa causar o programa exibir erroneamente cinco resumos.

5. Se a entrada ou saída de um programa for um conjunto ordenado (um arquivo sequencial, por exemplo, ou uma lista linear ou uma tabela), concentre a atenção no primeiro e no último elemento do conjunto.
6. Além disso, use sua criatividade para procurar outros limites
   condições.

O programa de análise de triângulos do Capítulo 1 pode ilustrar a necessidade de análise de valor de contorno. Para que os valores de entrada representem um triângulo, eles devem ser números inteiros maiores que 0, onde a soma de quaisquer dois é maior que o terceiro. Se você estivesse definindo partições equivalentes, você poderia definir uma onde esta condição é atendida e outra onde a soma de dois dos inte

gers não é maior que o terceiro. Portanto, dois casos de teste possíveis podem ser 3–4–5 e 1–2–4. No entanto, perdemos um erro provável. Isto é, se uma expressão no programa fosse codificada como AþB>¼C em vez de AþB>C, o programa nos diria erroneamente que 1–2–3 representa um triângulo escaleno válido. Portanto, a diferença importante entre a análise de valor de contorno e a partição de equivalência é que a análise de valor de contorno explora situações dentro e ao redor das bordas das partições de equivalência.

Como exemplo de uma análise de valor limite, considere a seguinte especificação de programa:

O MTEST é um programa que avalia exames de múltipla escolha. A entrada é um arquivo de dados chamado OCR, com vários registros com 80 caracteres. De acordo com a especificação do arquivo, o primeiro registro é um título usado como título em cada relatório de saída. O próximo conjunto de registros descreve as respostas corretas no exame. Esses registros contêm um "2" como o último caractere na coluna 80. No primeiro registro desse conjunto, o número de perguntas é listado nas colunas 1 a 3 (um valor de 1 a 999).

As colunas 10–59 contêm as respostas corretas para as questões 1–50 (qualquer caractere é válido como resposta). Os registros subsequentes contêm, nas colunas 10–59, as respostas corretas para as questões 51–100, 101–150 e assim por diante.

O terceiro conjunto de registros descreve as respostas de cada aluno; cada um desses registros contém um "3" na coluna 80. Para cada aluno, o primeiro registro contém o nome ou número do aluno nas colunas 1–

9 (qualquer caractere); as colunas 10–59 contêm as respostas do aluno para as questões 1–50. Se o teste tiver mais de 50 perguntas, os registros subsequentes do aluno conterão as respostas 51–100, 101–150 e assim por diante, nas colunas 10–59. O número máximo de alunos é 200. Os dados de entrada são ilustrados na Figura 4.4. Os quatro registros de saída são:

1. Um relatório, ordenado por identificador de aluno, mostrando a nota de cada aluno (porcentagem de acertos) e classificação.
2. Um relatório semelhante, mas classificado por grau.

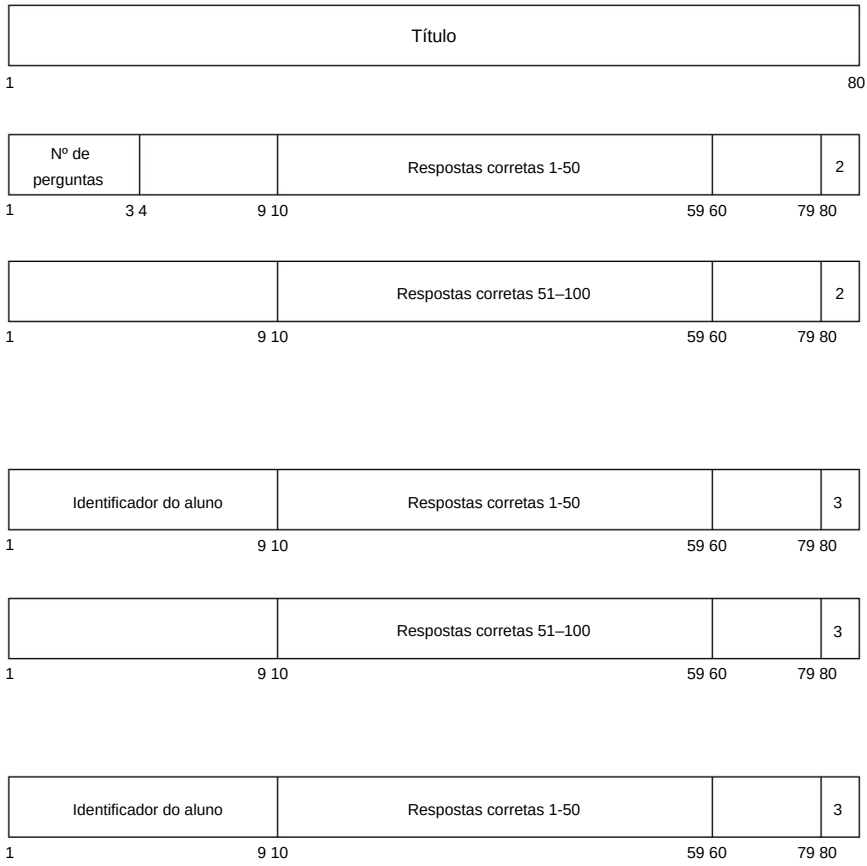

FIGURA 4.4 Entrada para o Programa MTEST.

3. Um relatório indicando a média, mediana e desvio padrão dos graus.
4. Um relatório, ordenado por número de pergunta, mostrando a porcentagem de idade dos alunos que responderam corretamente a cada pergunta.

Podemos começar lendo metodicamente a especificação, procurando condições de entrada. A primeira condição de entrada de limite é um arquivo de entrada vazio. A segunda condição de entrada é o registro de título; condições de contorno são registro de título ausente e os títulos mais curtos e mais longos possíveis. Nas próximas condições de entrada são a presença de registros de respostas corretas e a campo de número de perguntas no primeiro registro de resposta. A classe de equivalência

pois o número de perguntas não é de 1 a 999, porque algo especial acontece a cada múltiplo de 50 (ou seja, são necessários vários registros). Uma partição razoável disso em classes de equivalência é 1–50 e 51–999.

Portanto, precisamos de casos de teste em que o campo de número de perguntas seja definido como 0, 1, 50, 51 e 999. Isso cobre a maioria das condições de limite para o número de registros de respostas corretas; no entanto, três situações mais interessantes são a ausência de registros de respostas e ter um registro de resposta a mais e um a menos (por exemplo, o número de perguntas é 60, mas há três registros de resposta em um caso e um registro de resposta no outro caso).

Os casos de teste exclusivos identificados até agora são:

1. Arquivo de entrada vazio 2. Registro de título ausente 3. Título de 1 caractere 4. Título de 80 caracteres
5. Exame de 1 questão
6. Exame de 50 questões
7. Exame de 51 questões
8. Exame de 999 questões
9. Exame de 0 questões
10. Campo de número de questões com valor não numérico 11. Não há registros de respostas corretas após o título registro 12. Muitos registros de respostas corretas 13. Poucos registros de respostas corretas

As próximas condições de entrada estão relacionadas às respostas dos alunos. Os casos de teste de valor limite aqui parecem ser:

14. 0 alunos
15. 1 aluno
16. 200 alunos 17. 201 alunos 18. Um aluno tem um registro de resposta, mas há duas respostas corretas registros.
19. O aluno acima é o primeiro aluno do arquivo.
20. O aluno acima é o último aluno do arquivo.
21. Um aluno tem dois registros de respostas, mas há apenas um registro de respostas corretas.

22. O aluno acima é o primeiro aluno do arquivo.
23. O aluno acima é o último aluno do arquivo.

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

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

24. Todos os alunos recebem a mesma nota.
25. Todos os alunos recebem uma nota diferente.
26. Alguns alunos, mas não todos, recebem a mesma nota (para ver se as classificações são computadas corretamente).
27. Um aluno recebe uma nota 0.
28. Um aluno recebe uma nota 10.
29. Um aluno tem o menor valor de identificador possível (para verificar a classificação).
30. Um aluno tem o valor identificador mais alto possível.
31. O número de alunos é tal que o relatório é grande o suficiente para caber em uma página (para ver se uma página estranha é impressa).
32. O número de alunos é tal que todos os alunos, exceto um, cabem em um página.

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

33. A média está no máximo (todos os alunos têm nota perfeita).
34. A média é 0 (todos os alunos recebem nota 0).
35. O desvio padrão está no máximo (um aluno recebe um 0 e o outro recebe 100).
36. O desvio padrão é 0 (todos os alunos 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 isso é idêntico ao caso de teste 14.

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

37. Todos os alunos respondem corretamente à questão 1.
38. Todos os alunos respondem incorretamente à questão 1.
39. Todos os alunos respondem corretamente à última pergunta.
40. Todos os alunos respondem incorretamente à última pergunta.
41. O número de perguntas é tal que o relatório é grande o suficiente para caber em uma página.
42. O número de perguntas é tal que todas as perguntas, exceto uma, cabem em uma página.

Um programador experiente provavelmente concordaria neste ponto que muitos desses 42 casos de teste representam erros comuns que podem ter sido cometidos no desenvolvimento deste programa, mas a maioria desses erros provavelmente não seria detectada se um método de geração de casos de teste aleatório ou ad hoc fosse usado. A análise de valor limite, se praticada corretamente, é um dos métodos de projeto de caso de teste mais úteis. No entanto, muitas vezes é usado de forma ineficaz porque a técnica, na superfície, parece simples. Você deve entender que as condições de contorno podem ser muito sutis e, portanto, a identificação delas requer muita reflexão.

Representação gráfica de causa e efeito

Um ponto fraco da análise de valor limite e partição de equivalência é que eles não exploram combinações de circunstâncias de entrada. Por exemplo, talvez o programa MTEST da seção anterior falhe quando o produto do número de questões e o número de alunos excede algum limite (o programa fica sem memória, por exemplo). O teste de valor limite não detectaria necessariamente tal erro.

O teste de combinações de entrada não é uma tarefa simples porque mesmo se você particionar por equivalência as condições de entrada, o número de combinações geralmente é astronômico. Se você não tiver uma maneira sistemática de selecionar um subconjunto de condições de entrada, provavelmente selecionará um subconjunto arbitrário de condições, o que pode levar a um teste ineficaz.

A representação gráfica de causa-efeito auxilia na seleção, de forma sistemática, de um conjunto de casos de teste de alto rendimento. Tem um efeito colateral benéfico ao apontar incompletude e ambiguidades na especificação.

Um grafo causa-efeito é uma linguagem formal na qual uma linguagem natural especificação é traduzida. O gráfico na verdade é um circuito lógico digital (um rede lógica combinatória), mas em vez de notação eletrônica padrão, uma notação um pouco mais simples é usada. Nenhum conhecimento de eletrônica é necessário além da compreensão da lógica booleana (ou seja, dos operadores lógicos e, ou, e não).

O processo a seguir é usado para derivar casos de teste:

1. A especificação é dividida em partes viáveis. Isso é necessário porque os gráficos de causa e efeito tornam-se difíceis de manejar quando usados em especificações grandes. Por exemplo, ao testar um sistema de comércio eletrônico, uma solução viável peça pode ser a especificação para escolher e verificar um único item colocado em um carrinho de compras. Ao testar um design de página da Web, você pode teste uma única árvore de menu ou até mesmo uma sequência de navegação menos complexa.
2. As causas e efeitos na especificação são identificados. Uma causa é um 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 prolongado que uma entrada tem no estado do programa ou sistema). Por exemplo, se uma transação fizer com que um arquivo ou registro de banco de dados seja atualizada, a alteração é uma transformação do sistema; uma confirmação mensagem seria uma condição de saída.

   Você identifica causas e efeitos lendo a palavra de especificação por palavra e sublinhando palavras ou frases que descrevem causas e efeitos. Uma vez identificados, cada causa e efeito recebe um número único.
3. O conteúdo semântico da especificação é analisado e transformado em um gráfico booleano ligando as causas e os efeitos. Isto é o gráfico causa-efeito.
4. O gráfico é anotado com restrições que descrevem combinações de causas e/ou efeitos que são impossíveis por causa da sintaxe ou do ambiente restrições mentais.
5. Ao rastrear metodicamente as condições de estado no gráfico, você converte o gráfico em uma tabela de decisão de entrada limitada. Cada coluna do 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 um nó como tendo o valor 0 ou 1; 0 representa o estado "ausente" e 1 representa o estado "presente".

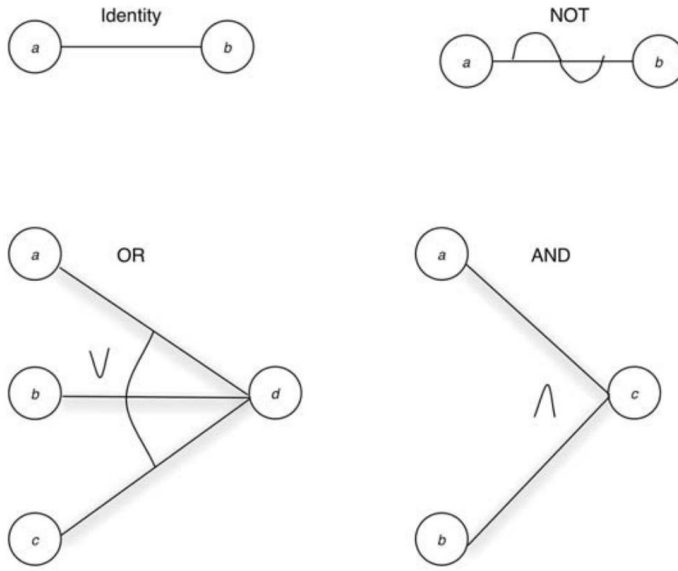

FIGURA 4.5 Símbolos básicos do gráfico de causa-efeito.

A função identidade afirma que se a é 1, b é 1; senão b é 0.
A função not afirma que se a é 1, b é 0, senão b é 1.
A função ou afirma que se a ou b ou c é 1, d é 1; senão d é 0.
A função and afirma que se a e b são 1, c é 1; senão 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 caractere na coluna 1 deve ser um "A" ou um "B". O caractere em
a coluna 2 deve ser um dígito. Nessa situação, a atualização do arquivo é feita. Se o primeiro caractere estiver incorreto, a mensagem X12 será emitida. Se o segundo caractere não for um dígito, a mensagem X13 é emitida.

As causas são:

1—caractere na coluna 1 é "A" 2—
caractere na coluna 1 é "B" 3—
caractere na coluna 2 é um dígito

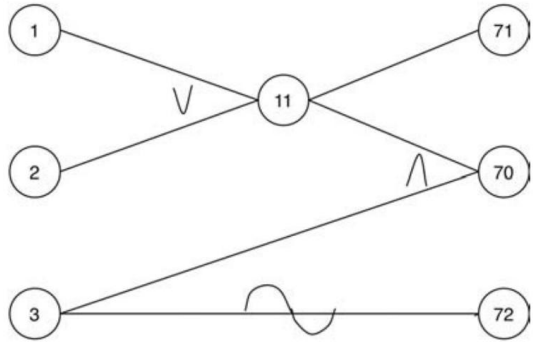

FIGURA 4.6 Exemplo de gráfico de causa-efeito.

e os efeitos são:

70—atualização feita
71—mensagem X12 é emitida
72—mensagem X13 é emitida

O gráfico causa-efeito é mostrado na Figura 4.6. Observe o nó intermediário 11 que foi criado. Você deve confirmar que o gráfico representa a especificação definindo todos os estados possíveis das causas e verificando se os efeitos estão definidos com os valores corretos. Para leitores familiarizados com diagramas lógicos, a Figura 4.7 é o circuito lógico equivalente.

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

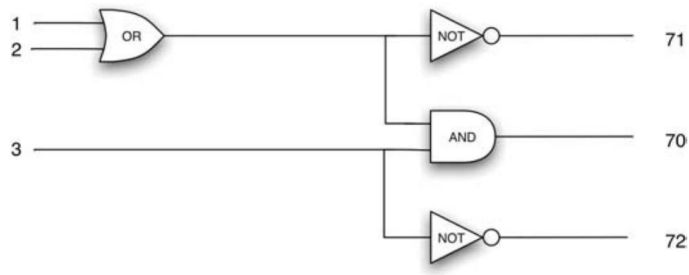

FIGURA 4.7 Diagrama Lógico Equivalente à Figura 4.6.

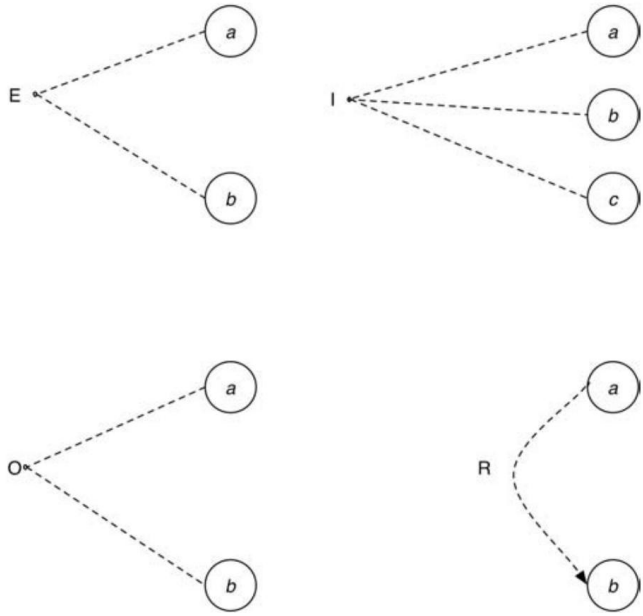

FIGURA 4.8 Símbolos de restrição.

Para explicar isso, a notação na Figura 4.8 é usada. 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 podem ser 0 simultaneamente).
A restrição O afirma que um, e apenas um, de a e b deve ser 1. A restrição R afirma que para a ser 1, b deve ser 1 (ou seja, é impossível que a seja 1 e b seja 0).

Frequentemente há a necessidade de uma restrição entre os efeitos. O M contra strain na Figura 4.9 afirma que se o efeito a é 1, o efeito b é forçado a 0.

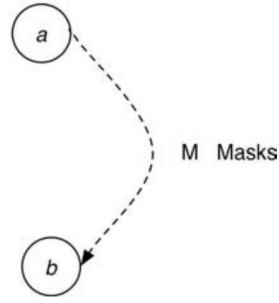

FIGURA 4.9 Símbolo para restrição "Máscaras".

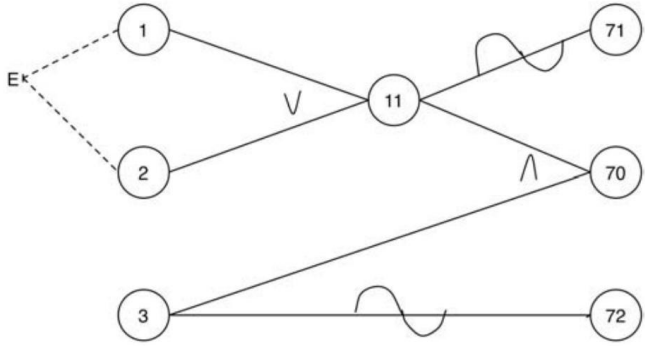

FIGURA 4.10 Exemplo de gráfico de causa-efeito com restrição "exclusiva".

Voltando ao exemplo simples anterior, vemos que é fisicamente impossível que as causas 1 e 2 estejam presentes simultaneamente, mas é possível que nenhuma delas esteja presente. Portanto, eles estão ligados à restrição E, conforme mostrado na Figura 4.10.

Para ilustrar como o gráfico de causa e efeito é usado para derivar casos de teste, use a seguinte especificação para um comando de depuração em um interativo sistema.

O comando DISPLAY é usado para visualizar a partir de uma janela de terminal o conteúdo dos locais de memória. A sintaxe do comando é mostrada em Figura 4.11. Os colchetes representam operandos opcionais alternativos. Letras maiúsculas representam palavras-chave de operandos. As letras minúsculas representam valores dos operandos (os valores reais devem ser substituídos). Os operandos sublinhados representam os valores padrão (ou seja, o valor usado quando o operando é omitido).

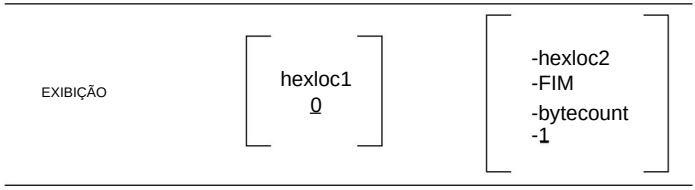

FIGURA 4.11 Sintaxe do Comando DISPLAY.

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, o endereço 0 é assumido. O endereço deve estar dentro do intervalo de memória real da máquina.

O segundo operando especifica a quantidade de memória a ser exibida. Se hexloc2 for especificado, ele define o endereço do último byte no intervalo de locais a serem exibidos. Pode ter de um a seis dígitos hexadecimais de comprimento. O endereço deve ser maior ou igual ao endereço inicial (hexloc1). Além disso, hexloc2 deve estar dentro do intervalo de memória real da máquina. Se END for especificado, a memória será exibida até o último byte real na máquina. Se for especificado bytecount , define o número de bytes de memória a serem exibidos (começando com o local especificado em hexloc1). O operando bytecount é um inteiro hexadecimal (um a seis dígitos). A soma de bytecount e hexloc1 não deve exceder o tamanho real da memória mais 1 e bytecount deve ter um valor de pelo menos 1.

Quando o conteúdo da memória é exibido, o formato de saída na tela é uma ou mais linhas do formato

xxxxxx ¼ palavra1 palavra2 palavra3 palavra4

onde xxxxxx é o endereço hexadecimal da palavra1. Um número inteiro de palavras (sequências de quatro bytes, onde o endereço do primeiro byte na palavra é um múltiplo de 4) é sempre exibido, independentemente do valor de hexloc1 ou da quantidade de memória a ser exibida. Todas as linhas de saída sempre conterão quatro palavras (16 bytes). O primeiro byte do intervalo exibido cairá na primeira palavra.

As mensagens de erro que podem ser produzidas são

M1 é uma sintaxe de comando inválida.
A memória M2 solicitada está além do limite de memória real.
A memória M3 solicitada é um intervalo zero ou negativo.

Como exemplos:

EXIBIÇÃO

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

MOSTRADOR 77F

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

MOSTRADOR 77F-407A

exibe as palavras que contêm os bytes no intervalo de endereços 775-407A;

MOSTRADOR 77F.6

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

EXIBIR 50FF-END

exibe as palavras contendo os bytes no intervalo de endereços 50FF até o final da memória.

O primeiro passo é uma análise cuidadosa da especificação para identificar os 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 faixa de memória real da máquina.
5. O segundo operando é END.
6. O segundo operando é 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 faixa de memória real da máquina.
12. O operando hexloc2 é maior ou igual ao operando hexloc1.
13. O operando bytecount contém apenas dígitos hexadecimais.
14. O operando bytecount contém de um a seis caracteres.

15. bytecount þ hexloc1 <¼ tamanho da memória þ
1. 16. bytecount >¼ 1.
17. A faixa especificada é grande o suficiente para exigir várias linhas de saída.
18. O início do intervalo não cai em um limite de palavra.

Cada causa recebeu um número único arbitrário. Observe que quatro causas (5 a 8) são necessárias para o segundo operando porque o segundo operando pode ser (1) END, (2) hexloc2, (3) byte-count, (4) ausente e (5) nenhum dos o de cima. 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 em uma linha.
95. A memória é exibida em várias linhas.
96. O primeiro byte do intervalo exibido cai em um limite de palavra.
97. O primeiro byte do intervalo exibido não cai em um limite de palavra.

O próximo passo 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 os 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 sintaticamente válido; o nó 35 representa um segundo operando sintaticamente válido. O nó 36 representa um comando sintaticamente válido. Se o nó 36 for 1, o efeito 91 (a mensagem de erro) não aparecerá. Se o nó 36 for 0, o efeito 91 está presente.

O gráfico completo é mostrado na Figura 4.13. Você deve explorá-lo com cuidado para se convencer de que ele reflete com precisão a especificação.

Se a Figura 4.13 fosse usada 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 por causa de restrições sintáticas. 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. Observe que, no máximo, uma das causas 5, 6, 7 e 8 pode estar presente. Todas as outras restrições de causa são a condição requer. Observe que a causa 17 (várias linhas de saída) requer o não da causa 8

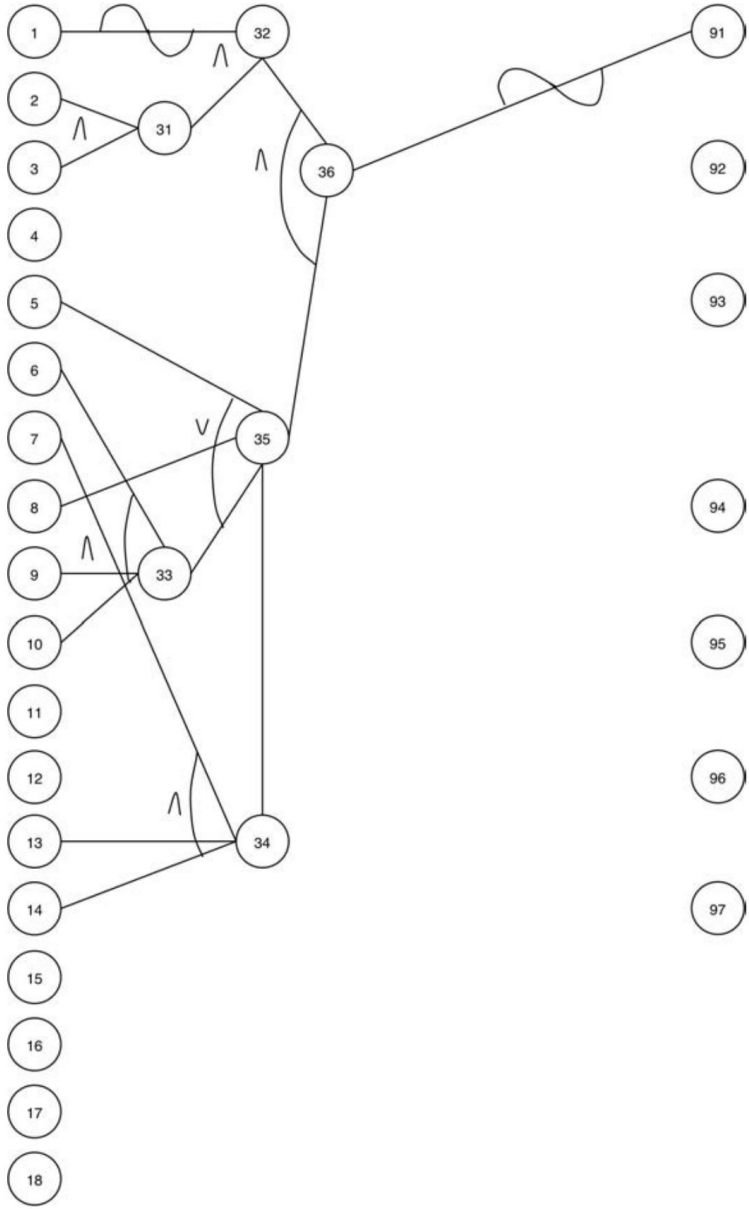

FIGURA 4.12 Início do Gráfico para o Comando DISPLAY.

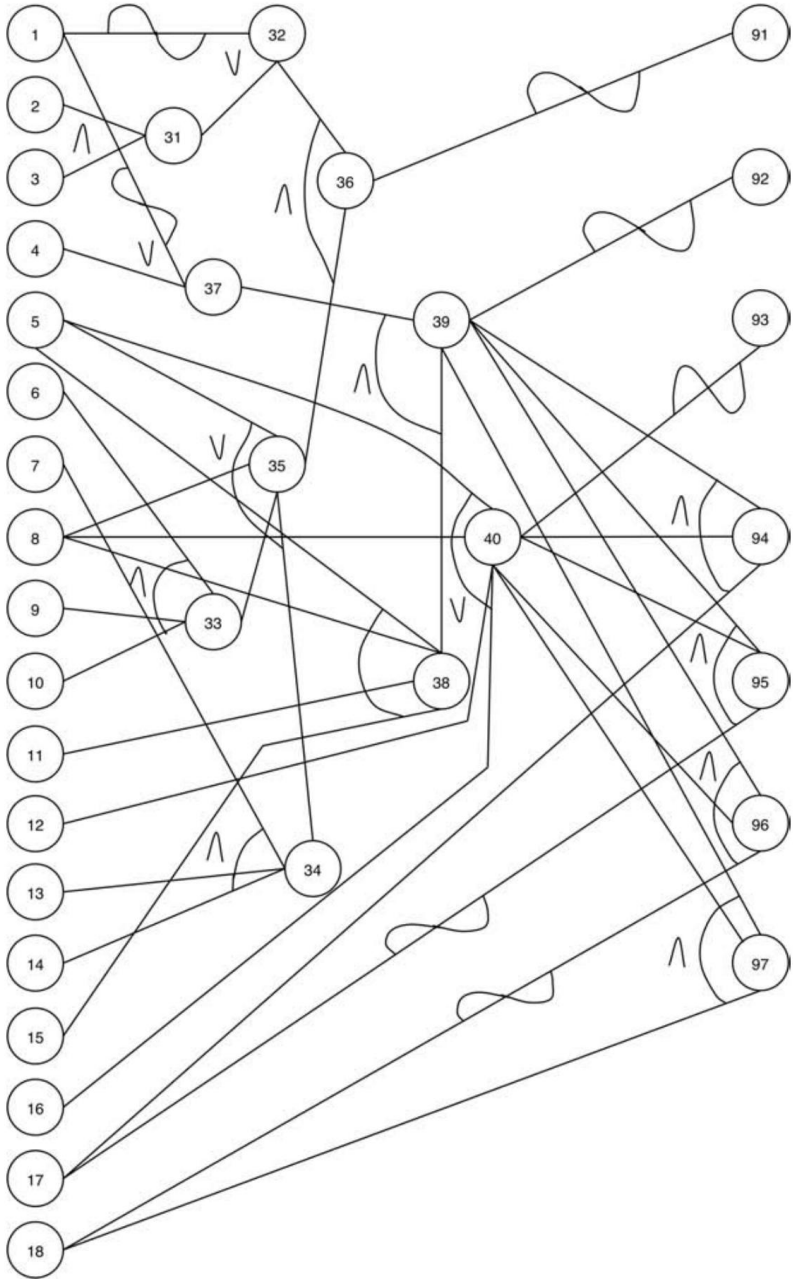

FIGURA 4.13 Gráfico completo de causa-efeito sem restrições.

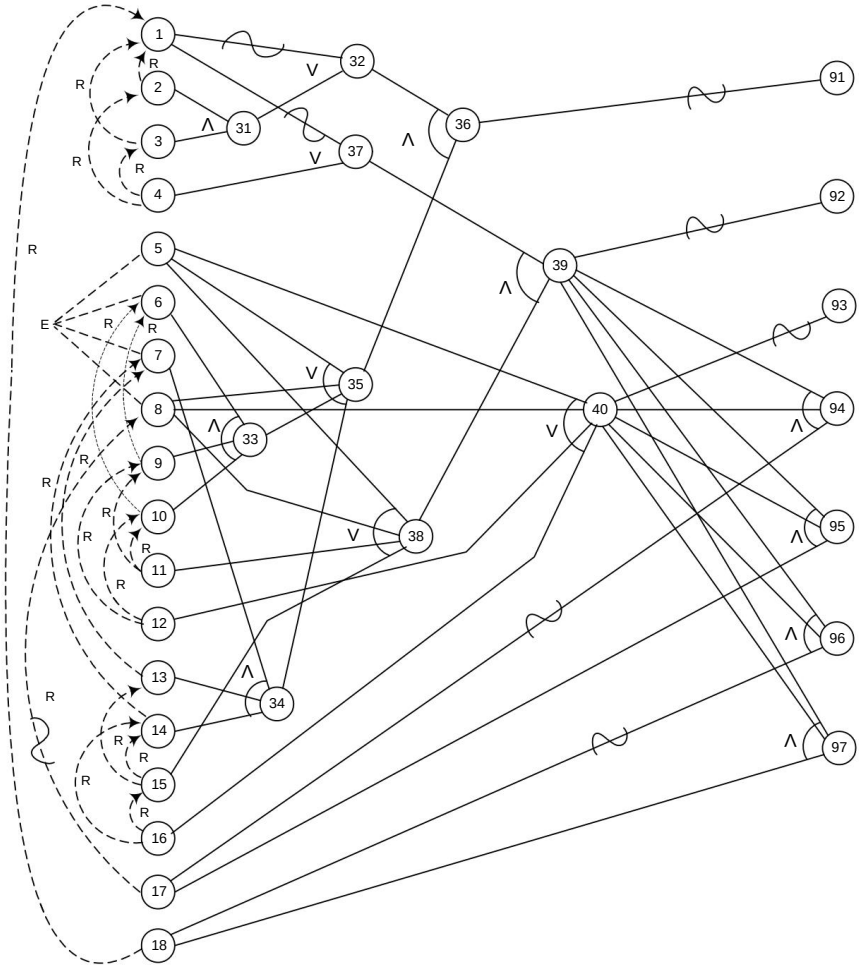

FIGURA 4.14 Gráfico completo de causa-efeito do comando DISPLAY.

(o segundo operando é omitido); a causa 17 pode estar presente somente quando a causa 8 está ausente. Novamente, você deve explorar as condições de restrição com cuidado.

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

1. Selecione um efeito para ser o estado atual (1).
2. Voltando ao gráfico, encontre todas as combinações de causas (sujeitas às restrições) que definirão esse efeito como 1.
3. Crie uma coluna na tabela de decisão para cada combinação de causas.

4. Para cada combinação, determine os estados de todos os outros efeitos e coloque-os em cada coluna.

Na execução da etapa 2, as considerações são as seguintes:

1. Ao rastrear de volta através de um nó ou cuja saída deve ser 1, nunca defina mais de uma entrada para ou como 1 simultaneamente. Isso é chamado de sensibilização de caminho. Seu objetivo é evitar a falha na detecção de certos erros por causa de uma causa mascarando outra causa.
2. Ao rastrear de volta através de um nó e cuja saída deve ser 0, todas as combinações de entradas que levam à saída 0 devem, é claro, ser enumeradas. No entanto, se você estiver explorando a situação em que uma entrada é 0 e uma ou mais das outras são 1, não é necessário enumerar todas as condições sob as quais as outras entradas podem ser 1.
3. Ao rastrear de volta através de um nó e cuja saída deve ser 0, apenas uma condição em que todas as entradas são zero precisa ser enumerada. (Se e está no meio do gráfico de tal forma que suas entradas vêm de outros nós intermediários, pode haver um número excessivamente grande de situações em que todas as suas entradas são 0.)

Essas considerações complicadas estão resumidas na Figura 4.15, e A Figura 4.16 é usada como exemplo.

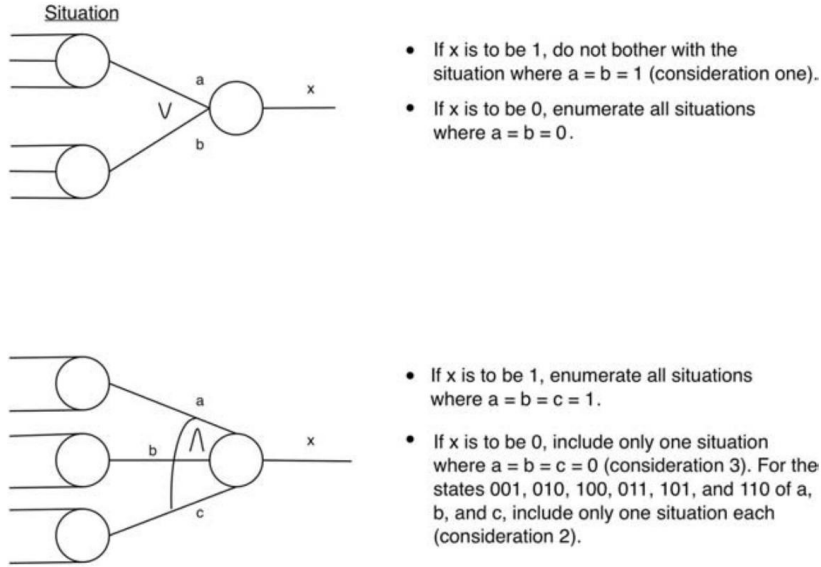

FIGURA 4.15 Considerações usadas ao traçar o gráfico.

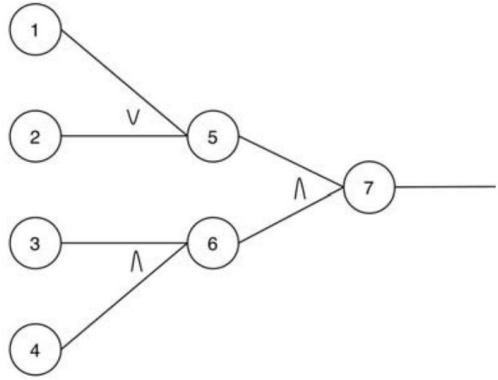

FIGURA 4.16 Exemplo de gráfico para ilustrar as considerações de rastreamento.

Suponha que queremos localizar todas as condições de entrada que causam a saída
state seja 0. A consideração 3 afirma que devemos listar apenas uma circunstância onde
os nós 5 e 6 são 0. A consideração 2 afirma que para o estado
onde o nó 5 é 1 e o nó 6 é 0, devemos listar apenas uma circunstância
onde o nó 5 é 1, em vez de enumerar todas as maneiras possíveis que o nó 5
pode ser 1. Da mesma forma, para o estado em que o nó 5 é 0 e o nó 6 é 1, temos
deve listar apenas uma circunstância em que o nó 6 é 1 (embora haja
apenas um 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. Daí, nós
chegaria a cinco estados dos 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 de nós 1 a 4 que levam a um 0
estado de saída.

Essas considerações podem parecer caprichosas, mas têm um propósito importante:
diminuir os efeitos combinados do gráfico. Eles eliminam situações que tendem a ser casos
de teste de baixo rendimento. Se casos de teste de baixo rendimento
não forem eliminados, um grande gráfico de causa e efeito produzirá um
número de casos de teste. Se o número de casos de teste for muito grande para ser prático,

você selecionará algum subconjunto, mas não há garantia de que o baixo rendimento casos de teste serão os eliminados. Por isso, é melhor eliminá-los durante a análise do gráfico.

Vamos agora converter o gráfico de causa-efeito da Figura 4.14 na tabela de decisão. O efeito 91 será selecionado primeiro. O efeito 91 está presente se o nó 36 estiver 0. O nó 36 é 0 se os nós 32 e 35 são 0,0; 0,1; ou 1,0; e considerações 2 e 3 aplicam-se aqui. Ao rastrear as causas e considerar as restrições entre as causas, você pode encontrar as combinações de causas que levam a efeito 91 estar presente, embora fazê-lo seja um processo trabalhoso.

A tabela de decisão resultante, sob a condição de que o efeito 91 esteja presente, é mostrado na Figura 4.17 (colunas 1 a 11). Colunas (testes) 1 a 3 representam as condições em que o nó 32 é 0 e o nó 35 é 1. Colunas 4 a 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 21 possíveis situações em que os nós 32 e 35 são 0 são identificadas. Os espaços em branco na tabela representam situações de “não me importo” (ou seja, o estado da causa é irrelevante) ou indicam que o estado de uma causa é óbvio por causa dos estados de outras causas dependentes (por exemplo, na coluna 1, sabemos que as causas 5, 7 e 8 devem ser 0 porque eles existem em uma situação de "no máximo um" com causa 6).

As colunas 12 a 15 representam as situações em que o efeito 92 está presente. As colunas 16 e 17 representam as situações em que o efeito 93 está presente. A Figura 4.18 representa o restante da tabela de decisão.

O último passo é converter a tabela de decisão em 38 casos de teste. Um conjunto de 38 casos de teste estão listados aqui. O número ou números ao lado de cada caso de teste designar os efeitos que se espera que estejam presentes. Suponha que o último localização na memória na máquina que está sendo usada é 7FFF.

1	EXIBIÇÃO 234AF74–123	(91)
2	MOSTRADOR 2ZX4–3000	(91)
3	MOSTRAR HHHHHHHH-2000	(91)
4	EXIBIR 200 200	(91)
5	DISPLAY 0–22222222	(91)
6	EXIBIR 1–2X	(91)
7	MOSTRADOR 2-ABCDEFGHI	(91)
8	MOSTRADOR 3.1111111	(91)
9	MOSTRAR 44.$42	(91)
10	EXIBIR 100.$$$$$$$$	(91)

	1	2	3	4	5	6	7	8	9	10	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					
3		0	1	0	1			1	1	1	0	1					
4												1	0	0	1	1	
5				0										1			
6	1	1	1	0	1	1					11				11		
7				0				1	1	1	1						
8				0													
9	1	1	1	1	0	0					01				11		
10	1	1	1	0	1	0					11				11		
11												0			01		
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	0	0	0	0	0		
92	0	0	0	0	0	0	0	0	1	1	1	1	0	0			
93	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				
95	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				
97	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	10		1	1	1	1	11		0	0	0	11		1
2	1	111							11	111	1			1					1	1	1
3	1	111							11	111	1			1					1	1	1
4	1	111							11	111	1			1					1	1	1
5	1				1				1	111											
6			1				1	1111													
7				1				1	1111												
8	111																				
9			1				1	1111													
10			1				1	1111													
11			1				1	1111													
12			1				1	1111													
13				1				1	1111												
14				1				1	1111												
15				1				1	1111												
16				1				1	1111												
17	0	000000	0	0	0001	1								1	111	1			1		1
18	1	111000	0	0	0001	1								1	000	0	0				0
91	0	000000	0	0	0000	0	0	000	0	0											0
92	0	000000	0	0	0000	0	0	000	0	0											0
93	0	000000	0	0	0000	0	0	000	0	0											0
94	1	111111	1						1	1110	0	0				0	0	0			0
95	0	000000	0	0	0001	1								1	111	1			1		1
96	0	000111	1						1	1110	0	0	111	1					1		1
97	1	111000	0	0	0001	1								1	000	0	0				0

FIGURA 4.18 Segunda Metade da Tabela de Decisão Resultante.

11	EXIBIR 10000000-M	(91)
12	MOSTRADOR FF-8000	(92)
13	EXIBIR FFF.7001	(92)
14	DISPLAY 8000-END	(92)
15	MOSTRADOR 8000–8001	(92)
16	VISOR AA-A9	(93)
17	EXIBIR 7.000,0	(93)
18	EXIBIÇÃO 7FF9-END	(94, 97)

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

Observe que onde dois ou mais casos de teste diferentes são invocados, para a maioria parte, o mesmo conjunto de causas, diferentes valores para as causas foram selecionados para melhorar ligeiramente o rendimento dos casos de teste. Observe também que, devido à tamanho real de armazenamento, caso de teste 22 é impossível (produzirá efeito 95 em vez de 94, conforme observado no caso de teste 33). Assim, 37 casos de teste foram identificados.

Comentários A representação gráfica de causa-efeito é um método sistemático de gerar casos representando combinações de condições. A alternativa seria fazer uma seleção ad hoc de combinações; mas ao fazê-lo, é provável que você ignoraria muitos dos casos de teste "interessantes" identificados pelo gráfico de causa e efeito.

Como o gráfico de causa e efeito requer a tradução de uma especificação em uma rede lógica booleana, ela oferece uma perspectiva diferente e uma visão adicional da especificação. Na verdade, o desenvolvimento de uma causa

gráfico de efeito é uma boa maneira 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.
Isso não pode ser verdade em todos os casos; ele não pode ocorrer para os casos de teste 18 e 26 porque o endereço inicial está a menos de 16 bytes do final da memória.

Embora o gráfico de causa e efeito produza um conjunto de casos de teste úteis, normalmente não produz todos os casos de teste úteis que podem ser identificados. Por exemplo, no exemplo não dissemos nada sobre verificar se os valores de memória exibidos são idênticos aos valores na memória e determinar se o programa pode exibir todos os valores possíveis em um local de memória. Além disso, o gráfico causa-efeito não explora adequadamente as condições de contorno. Claro, você pode tentar cobrir as condições de contorno durante o processo. Por exemplo, em vez de identificar a causa única

hexloc2>¼hexloc1

você pode identificar duas causas:

hexloc2 ¼ hexloc1
hexloc2 > hexloc1

O problema em fazer isso, no entanto, é que complica tremendamente o gráfico e leva a um número excessivamente grande de casos de teste. Por esta razão, é melhor considerar uma análise de valor de limite separada. Por exemplo, as seguintes condições de contorno 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. display 16
bytes (uma linha) 23. display 17
bytes (duas linhas)

Observe que isso não implica que você escreveria 60 (37 þ 23) casos de teste. Como o gráfico de causa-efeito nos dá margem de manobra na seleção de valores específicos para operandos, as condições de contorno podem ser combinadas nos casos de teste derivados do gráfico de causa-efeito. Neste exemplo, reescrevendo alguns dos 37 casos de teste originais, todas as 23 condições de contorno podem ser cobertas sem nenhum caso de teste adicional. Assim, chegamos a um conjunto pequeno, mas potente, de casos de teste que satisfazem ambos os objetivos.

Observe que o gráfico de causa e efeito é consistente com vários dos princípios de teste do Capítulo 2. A identificação da saída esperada de cada caso de teste é uma parte inerente da técnica (cada coluna na tabela de decisão indica os efeitos esperados). Observe também que isso nos encoraja a procurar efeitos colaterais indesejados. Por exemplo, a coluna (teste) 1 especifica que devemos esperar que o efeito 91 esteja presente e que os efeitos 92 a 97 estejam ausentes.

O aspecto mais difícil da técnica é a conversão do gráfico em tabela de decisão. Esse processo é algorítmico, o que implica que você pode automatizá-lo escrevendo um programa; existem vários programas comerciais para ajudar na conversão.

Erro ao adivinhar

Tem-se observado com frequência que algumas pessoas parecem ser naturalmente hábeis em testes de programas. Sem usar nenhuma metodologia específica, como limite

análise de valor de gráficos de causa e efeito, essas pessoas parecem ter um talento especial para farejar erros.

Uma explicação para isso é que essas pessoas estão praticando - subconscientemente com mais frequência do que não - uma técnica de projeto de caso de teste que poderia ser chamada de adivinhação de erros. Dado um programa específico, eles supõem – tanto por intuição quanto por experiência – certos tipos prováveis de erros e então escrevem casos de teste para expor esses erros.

É difícil fornecer um procedimento para a técnica de adivinhação de erros, uma vez que é em grande parte um processo intuitivo e ad hoc. A idéia básica é enumerar uma lista de possíveis erros ou situações propensas a erros e então escrever casos de teste com base na lista. Por exemplo, a presença do valor 0 na entrada de um programa é uma situação propensa a erros. Portanto, você pode escrever casos de teste para os quais valores de entrada específicos têm um valor 0 e para os quais valores de saída específicos são forçados a 0. Além disso, onde um número variável de entradas ou saídas pode estar presente (por exemplo, o número de entradas em um 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, fatores que foram omitidos da especificação, seja por acidente ou porque o redator achou que eram óbvios).

Uma vez que um procedimento para adivinhação de erros não pode ser dado, a próxima melhor alternativa é discutir o espírito da prática, e a melhor maneira de fazer isso é apresentando exemplos. Se você estiver testando uma sub-rotina de classificação, as seguintes situações são a serem exploradas:

A lista de entrada está vazia.
A lista de entrada contém uma entrada.
Todas as entradas na lista de entrada têm o mesmo valor.
A lista de entrada já está classificada.

Em outras palavras, você enumera os casos especiais que podem ter sido negligenciados quando o programa foi projetado. Se estiver testando uma sub-rotina de pesquisa binária, você pode tentar as situações em que: (1) há apenas uma entrada na tabela que está sendo pesquisada; (2) o tamanho da tabela é uma potência de 2 (por exemplo, 16); e (3) o tamanho da tabela é um menor e um maior que uma potência de 2 (por exemplo, 15 ou 17).

Considere o programa MTEST na seção sobre análise de valor limite. Os seguintes testes adicionais vêm à mente ao usar a técnica de adivinhação de erros:

O programa aceita "em branco" como resposta?
Um registro tipo 2 (resposta) aparece no conjunto de registros tipo 3 (aluno).

Um registro sem 2 ou 3 na última coluna aparece como diferente do registro inicial (título).
Dois alunos têm o mesmo nome ou número.

Como a mediana é calculada de forma diferente dependendo se há um número par ou ímpar de itens, teste o programa para um número par de alunos e um número ímpar de alunos.

O campo número de perguntas tem um valor negativo.

Testes de adivinhação de erros que vêm à mente para o comando DISPLAY do seção anterior são as seguintes:

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

A estratégia

As metodologias de projeto de casos de teste discutidas neste capítulo podem ser combinadas em uma estratégia geral. A razão para combiná-los já deve ser óbvia: cada um contribui com um conjunto específico de casos de teste úteis, mas nenhum deles 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, inicie com gráficos de causa e efeito.
2. Em qualquer caso, use a análise de valor limite. Lembre-se de 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, conforme observado na seção sobre gráficos de causa e efeito, muitas ou todas elas podem ser incorporadas aos testes de causa e efeito.

3. Identifique as classes de equivalência válidas e inválidas para a entrada e saída e complemente os casos de teste identificados acima, se necessário.
4. Use a técnica de adivinhação de erros para adicionar casos de teste adicionais.
5. Examine a lógica do programa em relação ao conjunto de casos de teste. Use o critério de cobertura de decisão, cobertura de condição, idade de cobertura de decisão/condição ou critério de cobertura de várias condições (sendo o último o mais completo). Se o critério de cobertura não tiver sido atendido pelos casos de teste identificados nas quatro etapas anteriores, e se atender ao critério não for impossível (ou seja, certas combinações de condições podem ser impossíveis de criar devido à natureza do programa), adicione casos de teste suficientes para fazer com que o critério seja satisfeito.

Novamente, o uso dessa estratégia não garante que todos os erros sejam encontrados, mas foi considerado um compromisso razoável. Além disso, representa uma quantidade considerável de trabalho árduo, mas, como dissemos no início deste capítulo, ninguém jamais afirmou que testar programas é fácil.

Resumo

Uma vez que você tenha concordado que o teste de software agressivo é uma adição valiosa aos seus esforços de desenvolvimento, a próxima etapa é projetar casos de teste que exercitarão seu aplicativo o suficiente para produzir resultados de teste satisfatórios. Na maioria dos casos, considere uma combinação de metodologias de caixa preta e caixa branca para garantir que você tenha projetado testes rigorosos de programa.

As técnicas de projeto de caso de teste discutidas neste capítulo incluem:

Cobertura lógica. Testes que exercitam todos os resultados do ponto de decisão pelo menos uma vez e garantem que todas as instruções ou pontos de entrada sejam executados pelo menos uma vez.

Particionamento equivalente. Define condições ou classes de erro para ajudar a reduzir o número de testes finitos. Assume que um teste de um valor representativo dentro de uma classe também testa todos os valores ou condições dentro dessa classe.

Análise de valor limite. Testa cada condição de aresta de uma classe de equivalência; também considera classes de equivalência de saída, bem como classes de entrada.

Gráficos de causa-efeito. Produz representações gráficas booleanas de resultados de casos de teste em potencial para auxiliar na seleção eficiente e completa
casos de teste.

> Erro ao adivinhar. Produz casos de teste com base no conhecimento intuitivo e especializado dos membros da equipe de teste para definir possíveis erros de software para facilitar o design eficiente do caso de teste.

Testes extensivos e aprofundados não são fáceis; nem o projeto de caso de teste mais extenso garantirá que todos os erros sejam descobertos. Dito isso, os desenvolvedores dispostos a ir além dos testes superficiais, que dedicarão tempo suficiente ao design do caso de teste, analisarão cuidadosamente os resultados do teste e agirão decisivamente sobre as descobertas, serão recompensados com um software funcional e confiável, razoavelmente livre de erros.