제 9 장 구조 기반 테스트

9.1 개 요

구조 기반 테스트(Structure-based test)는 프로그램 제어 흐름이나 자료 흐름 정보를 이용하여 테스트 케이스를 설계하는 방법이다. 구조 기반 테스트는 프로그램의 내부 구조 정보를 기반으로 테스트 케이스를 설계하는 측면에서 구조적 테스트(Structural test), 화이트박스 테스트(White box test) 또는 글래스 박스 테스트(Glass-box test)라고도 한다.

9.2 제어 흐름 그래프

제어 흐름 그래프(Control flow graph)는 프로그램 구조를 매우 효과적으로 나타낼 수 있는 수단이다. 구조 기반 테스트를 수행할 때 우선 프로그램을 제어 흐름 그래프로 나타낸 후에 테스트 케이스를 추출하는 것이 일반적이다. 제어 흐름 그래프를 이루는 요소는 다음과 같다:

- **기본 블록**(Basic block): 단일 진입점과 단일 진출점을 가진 일련의 연속적인 실행 가능한 문장들의 집합이다. 기본 블록의 문장들은 모두 함께 실행되거나 모두 함께 실행되지 않는다. 기본 블록은 각각 제어 흐름 그래프의 노드가 되며 박스로 표시한다.

- **제어 흐름**(Control flow): 기본 블록 간의 실행 순서를 나타낸다. 만약 기본 블록 A에서 기본 블록 B로 간선이 있다면 A가 실행된 후에 B가 실행된다는 의미이다. 제어 흐름은 화살표로 표시한다.

Exercise 01	그림 9.1 코드를 제어 흐름 그래프로 나타내 보자.

CFG.java

```
void findVinArray(int a[], int n, int v) {
        1: int i = 0;
        2: int count = 0;
        3: while (i < n) {
        4:      if (a[i] == v)
        5:                  count++;
        6:      i++;
           }
    7: printf("number %d", count);

  }
```

그림 9.1 예제 프로그램

표 9.1은 위 코드를 제어 흐름 그래프로 나타내기 위해 기본 블록들을 식별한 것이며 이를 제어 흐름 그래프로 나타낸 것이 그림 9.2이다.

표 9.1 기본 블록

기본 블록	문장	진입점	진출점
B1	{1, 2}	1	2
B2	{3}	3	3
B3	{4}	4	4
B4	{5}	5	5
B5	{6}	6	6
B6	{7}	7	7

기본 블록 B1은 1, 2번 문장들로 구성되어 있다. 3번 문장이 B1에 속하지 않는 이유는 무엇일까? 6번 문장의 실행이 종료되면 3번 문장으로 제어가 이동되어야 한다. 따라서 3번 문장은 1번 문장과 2번 문장과 동일한 블록에 포함될 수 없다. 또한, 3번 조건에 의해 조건이 참이 되면 4번 문장이 실행되나 거짓이 되면 4번 문장은 실행되지 않는다. 진출점이 두 개가 되기 때문에 3번과 4번 문장은 하나의 블록에 포함되지 않는다. 다른 기본 블록도 마찬가지로 구성된다.

그림 9.2는 기본 블록 간의 실행 순서를 고려하여 제어 흐름 그래프로 나타낸 것이다. 제어 흐름 그래프에서 볼 수 있듯이 프로그램 실행 흐름 및 프로그램 구조가 일목요연하게 나타나 있음을 알 수 있다.

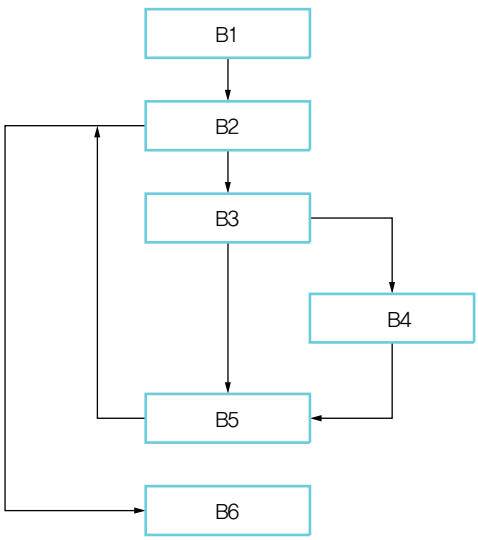

그림 9.2 제어 흐름 그래프 예

제어 흐름 그래프상에서 입력 간선이 없는 노드를 시작 노드라 하고, 출력 간선이 없는 노드를 종료 노드라 한다. 그림 9.2에서 시작 노드는 B1, 종료 노드는 B6이다. 제어 흐름 그래프상에서 프로그램 경로(Path)는 시작 노드에서 종료 노드까지 일련의 노드들이다. 예를 들어, (B1, B2, B3, B4, B5, B2, B3, B5, B2, B6)는 프로그램상에 존재하는 수많은 프로그램 경로 중의 하나이다.

9.3 구조 기반 테스트의 이해

가장 이상적인 구조 기반 테스트는 프로그램의 모든 경로를 최소한 한 번은 실행하여 테스트하는 것이다. 그러나 불행하게도 모든 프로그램 경로를 한 번씩 실행시키는 작업은 현실적으로 불가능하다. 그 이유는 프로그램 경로가 엄청나게 많이 존재할 수 있기 때문이다.

예를 들어, 표 9.1의 예제 프로그램에서 경로의 개수는 n이 0과 같거나 작은 작은 수일 때는 1, n이 1일 때는 2, n이 2일 때는 2^2, …, n이 20인 경우는 2^{20}에 달하는 프로그램 경로

가 존재하게 된다. 이러한 모든 프로그램 경로를 실행하기 위해서는 엄청나게 많은 시간과 노력이 요구된다.

따라서 구조 기반 테스트는 프로그램상에 존재하는 모든 가능한 경로를 테스트하는 대신 일부 경로만 테스트하는 방법을 사용한다. 대표적인 기본 경로 테스트인 문장 테스트, 분기 테스트, 조건 테스트, 다중 조건 테스트, MCDC 및 자료 흐름 테스트 등은 모든 프로그램 경로를 테스트 하지 않고 일부 경로만 선정하는 기준을 제공한다.

9.4 문장 테스트

문장 테스트(Statement test)는 프로그램의 모든(실행 가능한) 문장을 최소한 한 번은 행하도록 요구한다. 문장 테스트는 그림 9.3의 절차로 수행된다.

(1) 테스트 대상 프로그램에 해당하는 제어 흐름 그래프를 작성한다..
(2) 모든 실행 가능한 기본 블록들을 지나가는 프로그램 경로 집합을 식별한다.
(3) 프로그램 경로 집합에 있는 각 프로그램 경로에 대해 다음을 수행한다.
  A. 경로를 실행하는 입력 데이터를 식별한다.
  B. 명세 등에서 해당 입력에 대한 기대 출력(Expected output)을 식별한다.

그림 9.3 문장 테스트를 수행하는 절차

그림 9.3의 절차로 제어 흐름 그래프의 모든 블록이 수행되었다면 당연히 프로그램의 모든 문장이 수행되었음을 의미한다.

Exercise 02

다음 프로그램에 대해 문장 테스트에 따라 테스트 케이스를 설계해보자.

명세 함수 foo는 입력 x와 z 또는 y와 z가 양수이거나 x, y 값에 상관없이 z가 10보다 크면 10을 반환한다. 그 외의 경우는 0을 반환한다.

```
int foo(int x, int y, int z) {
        int w=0;
        if (x>0 || y>0)
                z=z+10;
        if (z>10)
                w=10;
        return w;
}
```

그림 9.4 예제 프로그램

우선 제어 흐름 그래프를 작성한다. 그림 9.5는 테스트 대상 프로그램의 제어 흐름 그래프이다. 이 제어 흐름 그래프에서 모든 블록을 실행할 수 있는 프로그램 경로 집합을 구한다. 이러한 프로그램 경로 집합은 유일하지 않으며 여러 개가 존재할 수 있다. 예를 들면, TS1={⟨B1, B2, B3, B5⟩, ⟨B1, B3, B4, B5⟩}이나 TS2={⟨B1, B2, B3, B4, B5⟩}는 모두 문장 테스트 요건을 만족하는 경로 집합들이다. 만약 TS1을 테스트할 경로 집합으로 선정하였다면 전체 4개의 프로그램 경로 i.g., {⟨B1, B2, B3, B4, B5⟩, ⟨B1, B2, B3, B5⟩, ⟨B1, B3, B4, B5⟩, ⟨B1, B3, B5⟩} 중에서 2개의 경로를 선정하여 테스트하는 것이고 TS2를 선정하였다면 4개의 경로 중에서 1개의 경로만 테스트하는 것이다.

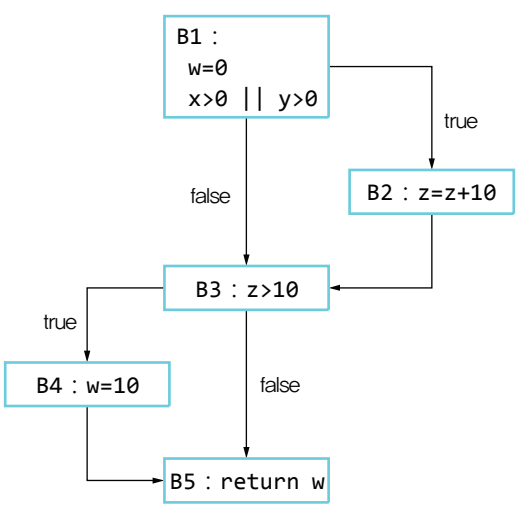

그림 9.5 제어 흐름 그래프

이 예에서는 TS2를 테스트 경로 집합으로 선정한다. 다음 단계에서는 TS2에 있는 경로를 실행하는 입력 데이터 집합을 구한다. 입력 데이터 집합 {(x=10, y=10, z=10)}은 프로그

램 경로 〈B1, B2, B3, B4, B5〉를 실행하므로 모든 프로그램 문장을 실행한다. 또한, 명세에서 이 입력 데이터가 주어졌을 때 기대 출력은 10임을 알 수 있다(표 9.2 참조).

표 9.2 테스트 케이스

테스트 케이스	입력			기대 출력	실행된 블록
	x	y	z		
1	10	10	10	10	B1, B2, B3, B4, B5

따라서 만약 프로그램에 (x=10, y=10, z=10)을 입력하여 실행했을 때 출력으로 10 이외의 값이 반환된다면 프로그램에 결함이 있음을 알 수 있다.

그림 9.6의 식을 이용하여 테스트 케이스 집합에 의해 문장 테스트가 어느 정도 이루어졌는지 정량적으로 알 수 있으며 이를 문장 커버리지(Coverage)라고 한다.

$$\text{문장 커버리지}(\%) = \frac{\text{테스트 케이스 집합에 의해 실행된 문장의 수}}{\text{전체 실행 가능한 프로그램 문장의 수}} \times 100$$

그림 9.6 문장 커버리지

Exercise 03

그림 9.5에서 (x=10, y=10, z=0)은 〈B1, B2, B3, B5〉 경로를 실행하기 때문에 문장 커버리지는 전체 6개의 문장 중에서 5개의 문장이 실행되므로 $\frac{5}{6} \times 100 = 83.3\%$의 문장 커버리지를 갖는다. 만약 실행이 안 된 블록 B4를 실행하는 테스트 케이스를 추가하면 100% 문장 커버리지를 가지게 될 것이다.

심화노트 단축연산과 문장 커버리지

C, C++나 Java와 같은 프로그래밍 언어는 프로그램의 효율적인 실행을 위해 단축연산(Short-circuit evaluation)을 수행한다. 단축 연산은 &&(and), ||(or)가 관여되는 논리 연산식에서 첫 번째 인자의 평가로 논리식의 결과가 결정될 때 다음에 나오는 인자를 평가하지 않는 방법이다. 만약 A&&B 연산식에서 A의 평가 결과가 false이면 B를 평가하지 않고 전체 연산의 결과는 false가 된다. 같은 이유로 A||B에서 A가 true이면 B를 평가하지 않고 전체 연산의 결과는 true가 된다.

만약 그림 9.4에서 "x>0 || y>0"가 단축 연산을 수행한다면 입력 데이터 집합 {(x=10), (y=10)}은 모든 문장을 실행하지 못한다. 그 이유는 논리식 "x>0 || y>0"에서 (x=10, y=10, z=10)에 대해 x>0가 참이 되어 전체 논리식의 결과를 결정하기 때문에 y>0이 평가(실행)되지 않기 때문이다. 이와 같이 단축 연산을 수행하는 경우에는 x>0의 결과를 false로 하여 y>0을 평가하게 할 필요가 있다. 따라서 (x=-10, y=10, z=10)을 입력하여 테스트하면 모든 문장을 실행할 수 있다.

9.5 결정 테스트

9.4의 문장 테스트(Statement test)는 더 적은 개수의 테스트 데이터들로 쉽게 만족할 수 있지만, 프로그램상에 존재하는 가능한 경우들을 모두 검증하지 못한다는 단점이 있다. 예를 들면, 다음 프로그램을 생각해보자.

```
if (x<0) x = -x;
z = x;
```

x가 0보다 작은 어떠한 값도 모든 문장을 한 번은 실행하게 하므로 문장 테스트의 요건을 만족한다. 이 경우에 변수 x가 0보다 크거나 같을 때는 프로그램이 올바르게 실행이 되는지 알 수가 없다.

결정 테스트는 프로그램상에 나타난 모든 결정문(Decision)의 결과가 참이 되는 경우와 거짓이 되는 경우를 최소한 한 번은 실행하도록 요구한다. 예를 들어, {(x=3), (x=−5)}는 결정문을 true와 false가 되는 경우를 실행하므로 결정 테스트 요건을 만족하는 테스트 데이터 집합이다. 물론 항상 결정문의 결과로 true와 false 2개의 값만을 가지지는 않는다. C 언어에서 스위치(Switch) 문장은 결정문이 여러 개의 결과를 가질 수 있다.

결정 테스트는 그림 9.7의 절차로 수행된다.

(1) 테스트 대상 프로그램에 해당하는 제어 흐름 그래프를 작성한다.
(2) 아직 실행되지 않은 결정의 결과(들)에 도달하는 프로그램 경로 집합을 식별한다.
(3) 프로그램 경로 집합에 있는 각 프로그램 경로에 다음을 수행한다.
  A. 경로를 실행하는 입력 데이터를 식별한다.
  B. 명세 등에서 해당 입력에 대한 기대 출력(Expected output)을 식별한다.
(4) (2)-(3)을 모든 결정의 결과가 실행될 때까지 반복한다.

그림 9.7 결정 테스트를 수행하는 절차

Exercise 04	Exercise 02의 프로그램을 사용하여 결정 테스트에 따라 테스트 케이스를 설계해보자.

이 프로그램에는 2개의 결정문이 있다: x > 0 || y > 0과 z > 10. 각각의 결정문이 true와

false 값을 갖는 프로그램 경로 집합을 구한다. 이러한 프로그램 경로 집합은 유일하지 않으며 여러 개가 존재할 수 있다.

예를 들어, TS={〈B1, B2, B3, B5〉, 〈B1, B3, B4, B5〉}는 그러한 집합 중의 하나이다. 경로 〈B1, B2, B3, B5〉를 실행하는 입력 테스트 데이터는 결정문 "x>0||y>0"이 true 값을 산출하고 z>10이 false 결과를 갖도록 한다. 경로 〈B1, B3, B4, B5〉를 실행하는 입력 테스트 데이터는 "x>0||y>0"이 false 값을 산출하고 z>10이 true 결과를 갖는다.

그림 9.8은 제어 흐름 그래프상에 이 경로 집합에 의해 실행된 경로들을 점선으로 표시한 것이다.

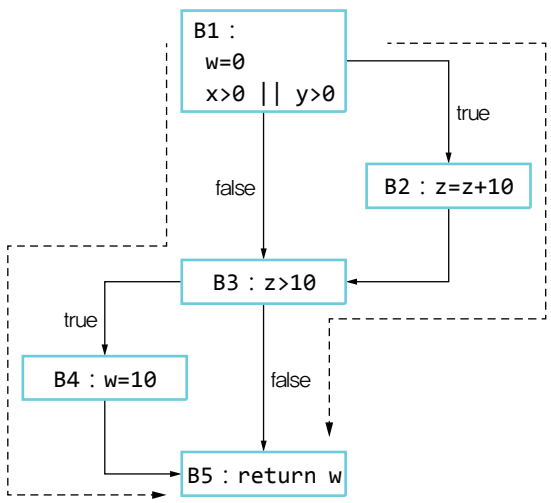

그림 9.8 실행된 경로

따라서 위 경로 집합을 실행하는 테스트 케이스 집합을 사용하여 프로그램을 실행하면 프로그램의 모든 결정문이 가질 수 있는 산출 결과들에 대해 프로그램을 테스트할 수 있다. 표 9.3은 결정 테스트에 따라 구한 테스트 케이스 집합을 보여 준다.

표 9.3 테스트 케이스

테스트 케이스	입력			기대 출력	결정 결과
	x	y	z		
1	10	10	−10	0	x>0||y>0: true
					z>10: false
2	−10	−10	20	10	x>0||y>0: false
					z>10: true

그림 9.9는 결정 커버리지를 구하는 식이다. 이 식을 이용하여 테스트 케이스 집합에 의해 결정 테스트가 어느 정도 이루어졌는지 정량적으로 알 수 있다.

$$\text{결정 커버리지}(\%) = \frac{\text{테스트 케이스 집합에 의해 실행된 결정문의 결과 수}}{\text{전체 프로그램의 결정문의 결과 수}} \times 100$$

그림 9.9 **결정 커버리지**

Exercise 05

표 9.3에서 테스트 케이스 1은 총 4개의 결정 결괏값 중에서 2개가 산출되기 때문에 $\frac{2}{4} \times 100 = 50\%$의 결정 커버리지를 갖는다. 여기에 테스트 케이스 2가 추가되면 100% 결정 커버리지를 갖는다.

심화노트 결정 테스트와 분기 테스트

ISO/IEC/IEEE 29119에서는 결정 테스트와 분기 테스트를 달리 정의한다. ISO/IEC/IEEE 29119에서 정의한 분기 테스트는 프로그램을 제어 흐름 그래프로 표현했을 때 제어 흐름 그래프상의 분기(또는 간선)들이 최소한 한 번은 실행되기를 요구한다. 이러한 정의에 따르면 표 9.3의 테스트 케이스 집합은 분기 테스트 요건을 만족한다.
그러나 결정문이 분기를 가지지 않은 경우에는 분기 테스트의 테스트 케이스 집합과 결정 테스트의 테스트 케이스 집합이 동일하지 않을 수 있다. 예를 들면, 다음 프로그램을 생각해보자.

```
int foo(int x, int y) {
      boolean k;
   1: k=(x>10)&&(y>10);
   2: return k+1;
}
```

1번 배정문은 결정식 "(x > 10)&&(y > 10)"의 결과를 변수 k에 대입한다. 따라서 결정 테스트로 테스트 케이스 집합을 구하면 k가 true와 false가 되는 경우를 모두 나오게 하는 x와 y의 값을 사용하여 테스트해야 한다. 즉, x와 y가 모두 10보다 큰 경우와 x나 y가 10보다 작은 경우인 두 개의 테스트 케이스가 필요하다. 반면에 분기 테스트는 제어 흐름 그래프를 작성했을 때 기본 블록 하나로만 구성되므로 이 블록을 실행하는 테스트 케이스이면 k가 어떤 값을 가지든지 상관없다. 따라서 하나의 테스트 케이스이면 분기 테스트의 요건을 만족한다.

테스트 요건 간의 관계는 포용 관계(Subsumption relation)로서 서로 비교할 수 있다. 만약 두 개의 테스트 요건이 C1과 C2가 있다고 가정할 때 C1을 만족하는 테스트 케이스 집합

이 C2를 만족한다면 C1은 C2를 포용한다고 말한다.

동일한 프로그램에 대해 결정 테스트를 만족하는 테스트 케이스 집합은 문장 테스트를 만족함을 쉽게 알 수 있다. 따라서 결정 테스트는 문장 테스트를 포용한다.

9.6 조건 테스트

DO-178B는 항공전자 시스템에서 사용되는 소프트웨어의 FAA(미국연방항공국) 승인을 위한 지침으로 널리 사용되어오고 있다. DO-178B는 다음과 같이 결정과 조건을 구분한다.

```
A Condition is a Boolean expression containing no Boolean operators. A Decision
is a Boolean expression composed of conditions and zero or more Boolean
operators. A decision without a Boolean operator is a condition. If a condition
appears more than once in a decision, each occurrence is a distinct condition.
```

위 정의에 따르면 조건은 논리 연산자 "and"나 "or" (C나 Java언어에서는 "&&", "||")를 포함하지 않은 Boolean 식이고, 결정은 이러한 조건들이 논리 연산자를 사용하여 구성된 Boolean 식을 의미한다. 그러나 기본적으로 우리는 DO-178B의 정의를 따르지만 조건이나 결정이 반드시 true나 false를 가지는 Boolean 식으로 한정하지 않는다. 그 이유는 switch 문과 같은 경우에는 true나 false가 아닌 여러 값을 가질 수 있기 때문이다.

예를 들어, 다음 프로그램을 생각해보자.

```
if (x>0 && y<=-3) {
        x=y+4;
        y = y-1;
}
```

이 프로그램에서 결정 "x>0&&y<=-3"은 두 개의 조건 x>0과 y<=-3을 논리 연산자 "&&"를 사용하여 구성하였다. 이때 x>0과 y<=-3은 식에 논리 연산자가 관여되지 않은, 더 이상 분할될 수 없는 (기본) 조건이다. 이와 같이 관계 연산자(=, <>, <=)만을 적용하거나 Boolean 변수로만 이루어진 식을 조건이라 한다.

이 프로그램에 결정 테스트 요건을 만족하는 테스트 집합을 구해보면 다음과 같다. {(x=4, y= −4), (x=−1, y=−4)}. 표 9.4에서 볼 수 있듯이 우리는 쉽게 위에서 주어진 테스트 집합을 사용하여 프로그램을 실행하면 결정이 true가 되는 경우와 false가 되는 경우를 모두 실행할 수 있음을 알 수 있다.

표 9.4 결정 테스트를 만족하는 테스트 집합

테스트 데이터	조건		결정
	x>0	y<=−3	x>0 &&y<=−3
x=4, y=−4	true	true	true
x=−1, y=−4	false	true	false

그러나 또한 이 테스트 집합은 조건 y<=−3이 false가 되는 경우는 전혀 테스트하지 못하는 사실도 알 수 있다.

조건 테스트(Condition test)는 프로그램의 조건에 나타난 모든 조건이 true가 되는 경우와 false가 되는 경우 모두를 발생하게 하는 입력 데이터를 테스트 집합으로 사용할 것을 요구한다. 따라서 표 9.4의 테스트 집합은 결정 테스트 요건을 만족하지만, y<=−3이 false가 되는 경우를 테스트하지 않기 때문에 조건 테스트를 만족하지 않는다. 만약 표 9.5와 같이 (x=−1, y=−2)를 테스트 집합에 추가하면 y<=−3이 false가 되는 경우를 테스트하므로 조건 테스트를 만족한다.

표 9.5 조건 테스트를 만족하는 테스트 집합

테스트 데이터	조건		결정
	x>0	y<=−3	x>0 &&y<=−3
x=4, y=−4	true	true	true
x=−1, y=−4	false	true	false
x=−1, y=−2	false	false	false

조건 테스트는 그림 9.10의 절차로 수행된다.

(1) 테스트 대상 프로그램에 해당하는 제어 흐름 그래프를 작성한다.
(2) 아직 실행되지 않은 조건의 결과(들)에 도달하는 프로그램 경로 집합을 식별한다.
(3) 프로그램 경로 집합에 있는 각 프로그램 경로에 다음을 수행한다.
  A. 경로를 실행하는 입력 데이터를 식별한다.
  B. 명세 등에서 해당 입력에 대한 기대 출력(Expected output)을 식별한다.
(4) (2)-(3)을 모든 조건의 결과가 실행될 때까지 반복한다.

그림 9.10 조건 테스트를 수행하는 절차

Exercise 06

Exercise 02의 프로그램을 사용하여 조건 테스트에 따라 테스트 케이스를 설계해보자. 이 프로그램에는 2개의 결정이 있다. 첫 번째 결정 x>0 || y>0에는 x>0과 y>0인 두 가지 조건으로 구성되어 있고, 두 번째 결정에는 z>10인 조건 하나로 이루어져 있다. 표 9.6은 조건 테스트를 만족하는 테스트 케이스 집합을 보여 준다.

표 9.6 테스트 케이스

테스트 케이스	입력			기대 출력	조건		
	x	y	z		x>0	y>0	z>10
1	1	−1	0	0	true	false	false
2	−1	1	1	10	false	true	true

그림 9.11의 식을 이용하여 테스트 케이스 집합에 의해 조건 테스트가 어느 정도 이루어졌는지 정량적으로 알 수 있으며, 이를 조건 커버리지 라고 한다.

$$\text{조건 커버리지}(\%) = \frac{\text{테스트 케이스 집합에 의해 실행된 개별 조건의 결과 수}}{\text{전체 프로그램 개별 조건의 결과 수}} \times 100$$

그림 9.11 조건 커버리지

Exercise 07

표 9.6에서 테스트 케이스 1은 총 6개의 결정 결괏값 중에서 3개가 산출되기 때문에 $\frac{3}{6} \times 100 = 50\%$의 조건 커버리지를 갖는다. 여기에 테스트 케이스 2가 추가되면 100% 조건 커버리지를 갖는다.

조건 테스트는 결정 테스트를 포용할까? 즉, 조건 테스트 요건을 만족하는 테스트 집합은 결정 테스트 요건을 만족할까? 대답은 No이다. 이는 표 9.7을 보면 쉽게 알 수 있다.

표 9.7 조건 테스트를 만족하지만 결정 테스트를 만족하지 않은 테스트 케이스

테스트 케이스	입력			기대 출력	조건		결정1	결정2
	x	y	z		x>0	y>0	x>0||y>0	z>10
1	1	−1	0	0	true	false	true	false
2	−1	1	1	10	false	true	true	true

표 9.7을 보면 2개의 테스트 케이스로 모든 조건의 결과가 생성됨을 볼 수 있지만, 결정1은 true만 테스트 되고 false는 테스트 되지 않는다. 따라서 표 9.7의 테스트 케이스 집합은 조건 커버리지를 만족하지만, 결정 커버리지를 만족하지 않는다. 또한, 우리는 결정 커버리지를 만족하지만 조건 커버리지가 반드시 만족되지 않는다는 사실을 이미 알고 있다. 따라서 조건 테스트와 결정 테스트는 서로 포용하지 않는다.

심화노트 단축 연산과 조건 커버리지

단축 연산을 수행한다면 표 9.7의 테스트 케이스 집합은 100% 조건 커버리지를 만족하지 못한다. 다음 표는 단축 연산을 수행하는 경우 표 9.7의 테스트 케이스 집합에 따른 조건들의 평가 결과를 보여 준다.

이 표에서 볼 수 있듯이 테스트 케이스 1은 x>0의 평가 결과로 전체 결정의 결과가 true가 됨을 알기 때문에 y>0 조건을 평가하지 않는다. 반면에 테스트 케이스 2는 x>0의 평가 결과가 false이기 때문에 결정의 결과를 알기 위해서는 y>0을 반드시 평가해야 한다.

테스트 케이스	입력			기대 출력	조건		결정1	결정2
	x	y	z		x>0	y>0	x>0||y>0	z>10
1	1	−1	0	0	true	–	true	false
2	−1	1	1	10	false	true	true	true

따라서 테스트 케이스 집합은 조건 y>0에서 false가 나오는 경우가 없으므로 100% 조건 커버리지를 만족하지 못한다. 100% 조건 커버리지를 만족하기 위해서는 y>0을 false가 되는 테스트 데이터를 추가할 필요가 있다. 이때 주의할 점은 x>0을 false가 나오도록 하면서 y>0을 false가 나오도록 해야 한다. 만약 x>0이 true가 나오면 y>0이 실행되지 않기 때문이다. 다음 표는 100% 조건 커버리지를 만족하도록 추가된 테스트 케이스를 보여 준다.

테스트 케이스	입력			기대 출력	조건		결정1	결정2
	x	y	z		x>0	y>0	x>0||y>0	z>10
1	1	−1	0	0	true	–	true	false
2	−1	1	1	10	false	true	true	true
3	−1	−1	1	0	false	false	false	false

9.7 결정/조건 테스트

9.6절에서 결정 테스트와 조건 테스트는 어느 쪽도 포용하지 않음을 기술하였다. 결정/조건 테스트(Decision Condition Test)는 결정 테스트와 조건 테스트를 모두 만족하는 테스트 케이스 집합을 설계하도록 요구한다.

결정 조건 테스트는 그림 9.12의 절차로 수행된다.

(1) 테스트 대상 프로그램에 해당하는 제어 흐름 그래프를 작성한다.
(2) 아직 실행되지 않은 결정과 조건의 결과(들)에 도달하는 프로그램 경로 집합을 식별한다.
(3) 프로그램 경로 집합에 있는 각 프로그램 경로에 다음을 수행한다.
  A. 경로를 실행하는 입력 데이터를 식별한다.
  B. 명세 등에서 해당 입력에 대한 기대 출력(Expected output)을 식별한다.
(4) (2)-(3)을 모든 결정과 조건의 결과가 실행될 때까지 반복한다.

그림 9.12 결정/조건 테스트를 수행하는 절차

Exercise 08

표 9.7을 보면 2개의 테스트 케이스로 모든 조건의 결과가 생성됨을 볼 수 있지만, 결정1은 true만 테스트 되고 false는 테스트 되지 않는다. 따라서 표 9.7의 테스트 케이스 집합은 조건 커버리지를 만족하지만, 결정 커버리지를 만족하지 않는다. 따라서 결정/조건 테스트의 요건을 만족하기 위해서는 결정1을 false로 평가할 수 있는 테스트 케이스가 추가되어야 한다. 표 9.8은 테스트 케이스 3을 추가하여 결정/조건 테스트를 만족하는 테스트 케이스 집합을 보여 준다.

표 9.8 결정 조건 테스트를 만족하는 테스트 케이스

테스트 케이스	입력			기대 출력	조건		결정1	결정2
	x	y	z		x>0	y>0	x>0||y>0	z>10
1	1	−1	0	0	true	false	true	false
2	−1	1	1	10	false	true	true	true
3	−1	−1	1	0	false	false	false	false

그림 9.13의 식을 이용하여 테스트 케이스 집합에 의해 결정 조건 테스트가 어느 정도 이루어졌는지 정량적으로 알 수 있으며, 이를 결정/조건 커버리지라고 한다.

$$\text{결정/조건 커버리지}(\%) = \frac{\text{테스트 케이스 집합에 의해 실행된 결정문과 개별 조건의 결과 수}}{\text{전체 프로그램 결정문과 개별 조건의 결과 수}} \times 100$$

그림 9.13 결정/조건 커버리지

Exercise 09

표 9.8의 테스트 케이스 1과 테스트 케이스 2로 이루어진 테스트 케이스 집합에 의해 달성되는 결정/조건 커버리지를 구해보자. 총 8개의 결정 및 조건 결괏값 중에서 7개가 산출되므로 $\frac{7}{8} \times 100 = 88\%$의 결정/조건 커버리지를 갖는다. 여기에 테스트 케이스 3을 추가하면 100% 결정/조건 커버리지를 갖는다.

Exercise 10

표 9.8의 테스트 케이스 집합은 결정/조건 커버리지를 만족하는 최소 집합이 아니다. 표 9.9와 같이 2개의 테스트 케이스를 사용하여 결정/조건 커버리지를 만족할 수 있다.

표 9.9 결정/조건 테스트를 만족하는 테스트 케이스

테스트 케이스	입력			기대 출력	조건		결정1	결정2
	x	y	z		x>0	y>0	x>0||y>0	z>10
1	1	1	0	0	true	true	true	false
2	−1	−1	1	20	false	false	false	true

9.8 다중 조건 테스트

더욱 강화된 테스트 케이스 집합을 얻기 위해서는 결정이 가질 수 있는 경우뿐만 아니라 결정을 구성하는 기본 조건들이 가질 수 있는 모든 가능한 조합까지도 프로그램 실행 중에 최소한 한 번은 검증할 필요가 있다.

예를 들어, 다음 프로그램을 생각해보자.

```
int multi(int x, int y) {
        if (x>0 && y <=-3) {
                x=y+4;
        }
        return x;
}
```

이 프로그램에서 결정 "x>0&&y<=-3"은 두 가지 조건 x>0과 y<=-3을 논리 연산자 "&&"를 사용하여 구성하였다. 다중 조건 테스트(Multiple Condition Test)는 표 9.10에서 보인 바와 같이 두 가지 조건의 모든 가능한 조합에 대해 테스트 집합을 구성한다.

표 9.10 다중 조건 테스트

id	x>0&&y<=-3		테스트 데이터
	x>0	y<=-3	
1	true	true	(x: 10, y: -5)
2	true	false	(x: 10, y: -1)
3	false	true	(x: -5, y: -5)
4	false	false	(x: -5, y: -1)

다중 조건 테스트(Multiple Condition Test)는 프로그램의 결정들에 사용된 모든 조건의 조합을 테스트할 수 있는 입력 데이터들을 테스트 데이터 집합으로 선정한다. 따라서 다중 조건 테스트는 문장 테스트, 결정 테스트, 조건 테스트 및 결정 조건 테스트를 포용한다.

다중 조건 테스트는 그림 9.14의 절차로 수행된다.

(1) 테스트 대상 프로그램에 해당하는 제어 흐름 그래프를 작성한다.
(2) 아직 실행되지 않은 조건의 조합을 실행하는 프로그램 경로들을 식별한다.
(3) 프로그램 경로 집합에 있는 각 프로그램 경로에 다음을 수행한다.
  A. 경로를 실행하는 입력 데이터를 식별한다.
  B. 명세 등에서 해당 입력에 대한 기대 출력(Expected output)을 식별한다.
(4) (2)-(3)을 모든 결정에 포함된 조건들의 조합이 실행될 때까지 반복한다.

그림 9.14 다중 조건 테스트를 수행하는 절차

Exercise 11

표 9.11은 Exercise 02의 프로그램에 다중 조건 테스트를 사용하여 구한 테스트 케이스 집합을 보여 준다. 이 프로그램에는 2개의 결정이 있고, 첫 번째 결정은 2개의 조건으로 구성되어 있으며 두 번째 결정은 하나의 조건으로 구성되어 있다. 테스트 케이스 1부터 테스트 케이스 4까지 첫 번째 결정을 구성하는 두 가지 조건의 모든 조합을 테스트한다. 또한, 테스트 케이스 1과 테스트 케이스 2는 두 번째 결정을 구성하는 모든 조건의 조합을 테스트함을 알 수 있다. 물론 테스트 케이스 1과 2 대신에 테스트 케이스 2와 3 또는 테스트 케이스 1과 4 또는 테스트 케이스 3과 4도 두 번째 결정을 구성하는 조건의 조합을 테스트한다.

표 9.11 다중 조건 테스트를 만족하는 테스트 케이스

테스트 케이스	입력			기대 출력	x>0||y>0		z>10
	x	y	z		x>0	y>0	z>10
1	1	1	1	10	true	true	true
2	1	0	0	0	true	false	false
3	−1	1	1	10	false	true	true
4	−1	−1	0	0	false	false	false

그림 9.15의 식을 이용하여 테스트 케이스 집합에 따라 다중 조건 테스트가 어느 정도 이루어졌는지 정량적으로 알 수 있으며 이를 다중 조건 커버리지 라고 한다.

$$\text{다중 조건 커버리지}(\%) = \frac{\text{테스트 케이스 집합에 의해 실행된 조건들의 조합 수}}{\text{전체 프로그램 개별 조건들의 조합 수}} \times 100$$

그림 9.15 다중 조건 커버리지

Exercise 12

표 9.11에서 테스트 케이스 1부터 테스트 케이스 3으로 구성된 3개의 테스트 케이스들에 의해 달성되는 다중 조건 커버리지를 구해보자.

총 6개의 조건 조합의 수(첫 번째 결정에서 4개의 조건 조합과 두 번째 결정에서 2개의 조건 조합)에서 첫 번째 결정의 조건 조합(false, false)이 누락되었기 때문에 5/6*100=84%의 다중조건 커버리지를 갖는다. 여기에 테스트 케이스 4를 추가하면 100% 다중 조건 커버리지를 갖는다.

실환노트 단축연산과 다중 조건 커버리지

단축 연산을 수행한다면 표 9.11의 테스트 케이스 1과 2는 x>0을 true로 평가하여 두 번째 결정 y>0을 실행하지 않는다. 따라서 첫 번째 결정의 관점에서 테스트 케이스 1과 2는 중복되는 테스트 케이스라 볼 수 있다. 테스트 케이스 3과 4는 모두 필요하다. 이 테스트 케이스들은 첫 번째 조건 x>0을 false로 평가하여 결국 두 번째 조건 y>0이 전체 결정의 값을 결정하기 때문이다. 두 번째 결정은 z>10인 한 가지 조건으로만 이루어져 있으므로 단축 연산의 영향을 받지 않으며, 이미 테스트 케이스 1과 4 또는 테스트 케이스 3과 4에 의해 다중 조건 커버리지를 만족한다.

테스트 케이스	입력			기대 출력	x>0||y>0		z>10
	x	y	z		x>0	y>0	z>10
1	1	1	1	10	true	–	true
3	−1	1	1	10	false	true	true
4	−1	−1	0	0	false	false	false