Más pruebas con JAVA

CoView determina los caminos linealmente independientes y los presenta en el ambiente de desarrollo coloreando el código.

En el artículo anterior utilizamos la herramienta CodeCover para conocer el cubrimiento alcanzado por un conjunto de casos de prueba a nivel de ciclos en el código. Este cubrimiento no brinda información sobre los caminos del código que se lograron recorrer al ejecutar las pruebas.
En este artículo presentamos el cubrimiento de Caminos Linealmente Independientes (CLI), sin entrar en detalles teóricos, y la herramienta CoView, que presenta visualmente cuáles de estos caminos fueron ejecutados durante las pruebas.

Caminos linealmente independientes

Un camino representa un flujo de ejecución desde el comienzo de un método (o programa) hasta su fin. Los posibles caminos dependen de las bifurcaciones existentes en el código (if, while, for, etc.) que dirigen el flujo de ejecución hacia "un lado u otro". Un método con N decisiones (if por ejemplo) y sin ciclos (loops) tiene potencialmente 2^N caminos. Si además tiene ciclos los caminos podrían ser infinitos. Desde el punto de vista del cubrimiento de código esta cantidad de caminos es normalmente imposible de cubrir o demasiado costosa.
Durante las pruebas, una forma de acotar la cantidad de caminos a cubrir es usar el criterio de CLI. El conjunto de CLI es el conjunto con menor cantidad de caminos que puede generar cualquier otro camino del método mediante combinación lineal de los mismos (esto es derivado del álgebra lineal). Es por este motivo que el criterio es interesante desde el punto de vista de las pruebas unitarias. Definir formalmente el conjunto de CLI queda fuera de este artículo.
Una forma de conocer la cantidad de CLI de un método es contar la cantidad de "zonas" en las cuales está dividido el grafo de flujo de control (grafo que indica los posibles caminos de un método). A este número se le llama la complejidad ciclomática del método.

Un ejemplo sencillo

Como ejemplo utilizamos el método merge que fue presentado en los artículos anteriores. En la Figura 1 se presenta el grafo de flujo de control de dicho método y se enumeran las "zonas" en la que está dividido el mismo; esto indica que el método tiene una complejidad ciclomática igual a 6. La zona numerada como 6 es la externa.

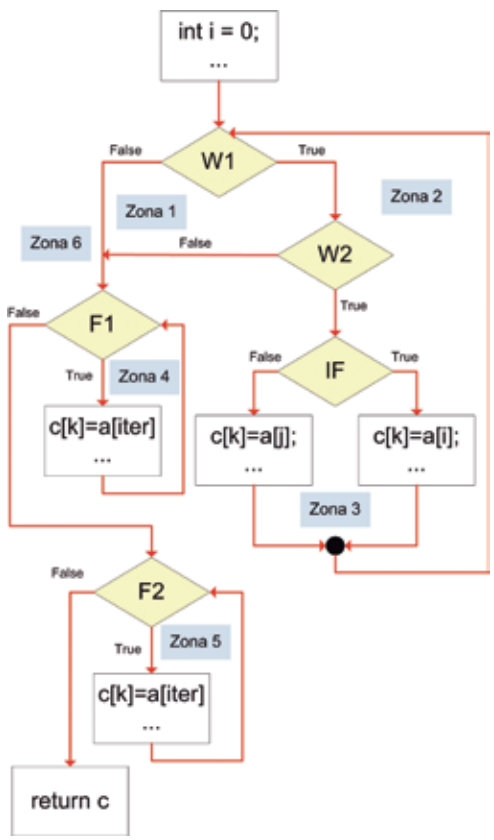

Figura 1 – Grafo de flujo de control del método merge

El grafo de flujo de control presenta al while del método merge dividido en dos (W1 y W2) donde cada W es una de las condiciones del while (W1 es i<a.length y W2 es j<b.length). F1 corresponde al primer for del código y F2 al segundo. Normalmente las bifurcaciones se representan con un rombo como aparece en la figura. Las sentencias secuenciales se representan con un rectángulo.

¿Cómo se determinan los caminos linealmente independientes?

Como vimos, la complejidad ciclomática indica la cantidad de caminos necesarios para obtener el conjunto de CLI pero no cuáles son esos caminos. Existen distintos métodos para obtener este conjunto, algunos son más complejos que otros. Solamente vamos a presentar el método utilizado por la herramienta CoView.

El método utilizado por CoView es el siguiente:

1. El primer camino a agregar al conjunto de CLI es el camino cuyas condiciones son todas True. Por ejemplo, con 5 condiciones como en nuestro ejemplo se representa este camino como el camino TTTTT.
2. Los otros caminos se derivan simplemente cambiando cada uno de esos valores en True por un valor en False. En nuestro caso son 5 caminos más.

El conjunto de CLI que deriva CoView en nuestro ejemplo es el siguiente {TTTTT, FTTTT, TFTTT, TTFTT, TTTFT, TTTTF}.

Analizando el conjunto de CLI con CoView

CoView determina los caminos linealmente independientes y los presenta en el ambiente de desarrollo coloreando el código. Dado un camino, las condiciones que se hacen True en ese camino son pintadas de verde y las que se hacen False de rojo. Las sentencias que no son condiciones se pintan de gris si son ejecutadas en el camino y si no son ejecutadas quedan en blanco.

La figura 2 contiene a la izquierda la forma en la cual CoView presenta el camino TTTTT y a la derecha el camino TTTFT. Ambas imágenes son usando CoView en el ambiente de desarrollo Eclipse.

```
public int[] mergearrays( int[ ] a, int [ ] b) {
    int i= 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while( i < a.length && j < b.length ){
        if( a[i] < b[j]){
            c[k]=a[i];
            k++;
            i++;
        }
        else{
            c[k]=b[j];
            k++;
            j++;
        }
    }
    for (int iter=i;iter<a.length;iter++){
        c[k]=a[iter];
        k++;
    }
    for (int iter=j;iter<b.length;iter++){
        c[k]=b[iter];
        k++;
    }
    return c;
}
```

```
public int[] mergearrays( int[ ] a, int [ ] b) {
    int i= 0;
    int j = 0;
    int k = 0;
    int c[] = new int[a.length + b.length];
    while( i < a.length && j < b.length ){
        if( a[i] < b[j]){
            c[k]=a[i];
            k++;
            i++;
        }
        else{
            c[k]=b[j];
            k++;
            j++;
        }
    }
    for (int iter=i;iter<a.length;iter++){
        c[k]=a[iter];
        k++;
    }
    for (int iter=j;iter<b.length;iter++){
        c[k]=b[iter];
        k++;
    }
    return c;
}
```

Figura 2 – Dos caminos del conjunto de CLI que determina CoView

Ejecución usando CoView

Utilizamos los mismos cuatro casos de prueba que utilizamos en el artículo anterior de forma de probar el método merge.
Al ejecutar estos casos de prueba CoView indica que solamente 1 de los 6 caminos del conjunto CLI ha sido ejecutado. Dado este resultado, y con el objetivo de alcanzar un cubrimiento del 100%, corresponde generar casos de prueba para cubrir los caminos del conjunto CLI que no se han cubierto.
Para generar estos casos de prueba analizamos los 6 caminos generados por CoView. Surge de este análisis que varios de estos caminos no son ejecutables. Es decir, que ningún caso de prueba durante su ejecución puede recorrer el grafo de flujo de control como lo indica el camino.

Como ejemplo tomemos el primer camino de la Figura 2 (TTTTT). Este camino indica que se entra al while con i<a.length y con j<b.length (el camino indica que el while se hace true). Luego, para cumplir con el camino, a[i] debe ser menor a b[j] (el if se hace true). Al retornar a la decisión del while no se tiene que ejecutar el cuerpo por lo que alguna de las dos condiciones es falsa; entonces, o i no es menor a a.length o j no es menor a b.length. Esto indica que al menos uno de los dos for del final no podrá ser ejecutado. De esta forma se muestra que el camino TTTTT no puede ser ejecutado con ningún caso de prueba ya que nunca va a suceder que ambos for ejecuten su cuerpo.

Analizando caminos no ejecutables

Como el conjunto de CLI determinado por CoView contiene caminos no ejecutables no se podrá obtener el 100% de cubrimiento. Sin embargo, no cumplir con el cubrimiento se transforma en una preocupación secundaria. ¿Por qué en el código hay caminos no ejecutables? ¿Existe alguna forma de solucionarlo? En nuestro próximo artículo presentaremos un nuevo algoritmo para merge que resuelve el problema de tener tantos caminos no ejecutables y estudiaremos el cubrimiento alcanzado con diversos criterios, incluyendo el criterio de CLI.

En este artículo vimos como usando el cubrimiento de CLI llegamos a detectar que el método merge tiene varios caminos no ejecutables. El descubrir caminos no ejecutables es un buen motivo para pensar una nueva solución algorítmica.
Como final debemos dejar en claro dos cosas. Primero, la utilización del criterio CLI no es necesaria para el análisis de caminos no ejecutables de un algoritmo; esta se puede hacer simplemente analizando el grafo de flujo de control directamente. Segundo, no siempre se puede conseguir un algoritmo sin caminos no ejecutables.

Diego Vallespir
Fernando Marotta

Grupo de Ingeniería de Software, UdelaR
gris@fing.edu.uy