

Priorización de casos de prueba mediante mutación

Macario Polo, Ignacio García-Rodríguez y Mario Piattini
Escuela Superior de Informática – Universidad de Castilla-La Mancha – España
{Macario.Polo, Ignacio.GRodriguez, Mario.Piattini}@uclm.es

Resumen

Se presenta un algoritmo para priorización de casos de prueba y una herramienta que lo implementa. El algoritmo obtiene altas reducciones en el tamaño del suite original sin pérdida de calidad. El criterio de selección de casos es el porcentaje de mutantes que mata cada caso de prueba.

1. Introducción

La priorización de casos de prueba es una práctica utilizada para disminuir los costes de las pruebas de regresión: básicamente, consiste en reejecutar aquellos casos que, de acuerdo con algún criterio de calidad, resultan más importante. En los últimos años, las organizaciones han adoptado los entornos X-Unit como herramientas para automatizar sus procesos de pruebas, lo que está permitiendo aplicar a nivel industrial los resultados obtenidos durante años de investigación en el plano académico.

A partir de diferentes *surveys*, en un artículo reciente hemos discutido el grado actual de automatización en compañías de desarrollo de software [1]: el estudio que muestra mejores resultados es el de Ng et al. en Australia [2], en donde el 79.5% de las compañías encuestadas automatizan la ejecución de pruebas y el 75% las de regresión; además, 38 de las organizaciones consultadas (58.5%) utilizan métricas para pruebas, siendo el número de defectos la más utilizada (31 organizaciones). Como es bien sabido, los entornos X-Unit permiten automatizar la ejecución de pruebas y las de regresión, y el principal resultado que muestran es el número de casos de prueba que encuentran fallo. Por tanto, y aunque el nivel de detalle de los trabajos consultados no es más fino, parece muy probable que las herramientas utilizadas por las compañías mencionadas sean del tipo X-Unit. De hecho, Do et al. afirman que JUnit está siendo

cada vez más utilizada por las compañías de desarrollo de software [3].

Este artículo describe una herramienta que prioriza casos de prueba de JUnit en función del número de mutantes muertos por los mismos casos, en formato MuJava [4]. La herramienta puede ejecutarse en modo *standalone* o desde *testooj*, una herramienta de automatización de las pruebas de programas Java [1, 5].

2. Trabajos relacionados

En esta sección se resumen algunos trabajos relacionados con este artículo. Dependiendo de la estrategia de generación de casos utilizada, *testooj* puede generar un número de casos demasiado grande, por lo que se desarrolló e implementó el algoritmo que se presenta como parte principal de este artículo. Previamente, en la siguiente subsección se describen algunos algoritmos relacionados con la reducción del conjunto de casos de prueba.

2.1. Algoritmos para la reducción del conjunto de casos de prueba

El problema de la “reducción óptima del *test-suite*”, tal y como lo enuncian Jones y Harrold [6], es el siguiente:

Dado: Un *Test Suite* T , un conjunto de requisitos r_1, r_2, \dots, r_n que deben ser satisfechos por los casos de prueba en relación a la cobertura de un programa,

Problema: Encontrar $T' \subseteq T$ tal que T' satisface todos los requisitos r_i y $(\forall T' \subseteq T, T'$ satisface r $\mathcal{P} \rightarrow |T'| \leq |T''|$)

En otras palabras, el problema consiste en encontrar, a partir de un conjunto T de casos de prueba, un subconjunto T'' de casos de prueba de cardinal mínimo que consiga la misma cobertura que T . A continuación revisamos algunos algoritmos.

2.1.1. Algoritmo HGS

Harrold, Gupta y Sofa [7] presentan un algoritmo voraz (referido en la literatura como algoritmo *HGS*) para reducir el tamaño del *test-suite*, mientras que se preservan los requisitos de prueba del original. El algoritmo es aplicable para varios requisitos de prueba (p.ej., obtener los mejores casos en cuanto a cobertura de todos los usos y de condición-decisión).

2.1.2. Algoritmo de Heimdahl y George

Heimdahl y George [8] proponen también un algoritmo voraz para reducir el *test-suite*: básicamente, toman un caso de prueba al azar, lo ejecutan y miden la cobertura alcanzada. Si ésta es mayor que la del caso que alcanzaba más cobertura, lo añaden al conjunto reducido. El algoritmo lo ejecutan cinco veces para obtener cinco conjuntos reducidos distintos. Ya que se confía completamente en el azar, la calidad de los resultados no está garantizada.

2.1.3. Algoritmo de McMaster y Memon

McMaster y Memon [9] presentan otro algoritmo voraz. El criterio para seleccionar casos de prueba es el número de llamadas a la pila que realice el programa bajo prueba ante el caso de prueba. Como se observa, éste no es un requisito de pruebas demasiado común.

2.1.4. Resumen

El problema descrito es NP-completo y su solución, por tanto, no puede ser obtenida en tiempo polinomial. Así, todos los algoritmos propuestos obtienen soluciones próximas a la óptima: el tamaño del *test-suite* se reduce manteniendo la calidad, pero no hay garantía de que el tamaño conseguido sea el mínimo posible.

El criterio de selección de casos puede ser cualquiera (cobertura de bloques, de sentencias, de condiciones...) o, por ejemplo, y como se presenta en este artículo, el número de mutantes que mata cada caso de prueba.

El algoritmo que se presenta en este artículo también es voraz y, así, garantiza que la cobertura del conjunto reducido es la misma que la del original. La principal diferencia con respecto a otros trabajos es el criterio de selección de casos (número de mutantes muertos) y el formato de los casos, que es compatible con los entornos JUnit.

2.2. Estrategias de combinación para generación de casos de prueba

En un artículo reciente, Grindal et al. describen 16 estrategias para generación de casos de prueba que clasifican en diversas categorías [10]: partiendo del conjunto de valores de prueba, el objetivo de estas estrategias es obtener Buenos conjuntos de casos de prueba que logren alta cobertura en el programa que se está probando.

De la estrategia seleccionada depende el tamaño del *test-suite*, como también la cobertura alcanzada en la clase bajo prueba. A modo de ejemplo, la Tabla 1 muestra el número de casos de prueba generados con tres de las estrategias revisadas, así como el porcentaje de mutantes muertos al ejecutarlos sobre una clase *Triangle*, que representa el clásico problema de determinación del tipo de un triángulo. Como se ve, *All combinations* construye el *test-suite* de mayor tamaño (216 casos), aunque también es la estrategia que obtiene mayor cobertura (medida, en este caso, como el porcentaje de mutantes muertos). En efecto, *All combinations* se utiliza habitualmente como estrategia de base para comparar el número y calidad de los casos generados por las técnicas que los investigadores van proponiendo.

Estrategia	Nº de casos	% muertos
Each choice	7	51%
Anti random	8	65%
All combinations	216	100%

Tabla 1. Resultados obtenidos con varias estrategias en el problema del triángulo

3. Reducción del conjunto de casos de prueba basado en mutación

En esta sección se describe un algoritmo voraz que utiliza el porcentaje de mutantes muertos como criterio para incluir casos de prueba en el conjunto reducido. La implementación dada al algoritmo permite seleccionar casos de prueba JUnit, si bien la selección se realice en función del porcentaje de mutantes que matan los mismos casos, en formato MuJava.

La equivalencia entre los casos de prueba JUnit y MuJava de uno a otro formato se discute y describe en la comunicación [5], presentada el año pasado en este mismo foro y en [1]; la Figura 1 ilustra la correspondencia entre uno y otro formato

con un sencillo ejemplo. Siendo t_j y t_m dos casos de prueba equivalentes en formatos JUnit y MuJava, el algoritmo selecciona t_j basándose en los mutantes muertos por t_m .

```

public void test1() {
    Account o=new Account();
    o.deposit(1000);
    assertTrue(o.getBalance()==1000);
}

public String test1() {
    Account o=new Account();
    o.deposit(1000);
    return o.toString();
}
    
```

Figura 1. Dos casos de prueba equivalentes en formatos JUnit y MuJava

La Figura 2 muestra la función principal del algoritmo. Como entradas, recibe el conjunto completo de casos de prueba, la clase bajo prueba y el conjunto completo de mutantes. En la línea 2, ejecuta todos los casos contra la clase y contra los mutantes, guardando los resultados en la variable *testCaseResults*.

```

1. reduceTestSuite(completeTC : SetOfTestCases,
   cut : CUT, mutants : SetOfMutants)
   : SetOfTestCases
2. testCaseResults = execute(completeTC, cut,
   mutants)
3. requiredTC = ∅
4. n=|mutants|
5. while (n>0)
6.   mutantsNowKilled = ∅
7.   testCasesThatKillN =
   getTestCasesThatKillN(completeTC, n, mutants,
   mutantsNowKilled, testCaseResults)
8.   if |testCasesThatKillN|>0 then
9.     requiredTC = requiredTC ∪ testCasesThatKillN
10.    for i=1 to |testCasesThatKillN|
11.      testCase = testCasesThatKillN[i]
12.      testCase.removeAllTheMutantsItKills()
13.    next
14.    n = |mutants|-|mutantsNowKilled|
15.  else
16.    n = n - 1
17.  end if
18.end_while
19.return requiredTC
20.end
    
```

Figura 2. Función principal del algoritmo, que devuelve el *test-suite* reducido

Entonces, el algoritmo está preparado para seleccionar, en varias iteraciones, los casos de prueba que matan más mutantes (líneas 5-18).

La primera vez que el algoritmo entra en este bucle y llega a la línea 7, el valor de n (utilizado para dejar de iterar) es $|mutants|$ (cardinal del conjunto de mutantes): en este caso especial, el algoritmo busca algún caso de prueba que mate a todos; si lo encuentra, el algoritmo añade el caso a *requiredTC*, actualice el valor de n y para; en otro caso, decrementa n (línea 16) y se introduce nuevamente en el bucle.

Supongamos que n es inicialmente 100 (es decir, hay 100 mutantes de la clase bajo prueba), y supongamos que el algoritmo no encuentra casos de prueba que maten a n mutantes hasta que $n=30$. Con este valor, la función *getTestCasesThatKillN* (llamada en la línea 7) devuelve todos los casos de prueba que maten a n mutantes diferentes: así, si dos casos (tc_1 y tc_2) matan los mismos 30 mutantes, *getTestCasesThatKillN* devuelve solo un caso (por ejemplo, tc_1). Si la intersección de los mutantes muertos por tc_1 y tc_2 no es vacía, entonces el algoritmo devuelve un conjunto formado por tc_1 y tc_2 .

Cuando se encuentran los casos que matan a n mutantes, se añaden a *requiredTC* (línea 9) y se eliminan los mutantes muertos del conjunto de mutantes (líneas 10-13). El valor de n se actualiza al número de mutantes que quedan vivos.

En la implementación real del algoritmo, la ejecución del conjunto completo de casos contra la clase bajo prueba y sus mutantes se realice en una función separada (función *execute*, llamada en la línea 2). Esta función devuelve un conjunto de objetos de tipo *TestCaseResult*, compuestos por el nombre del caso de prueba y la lista de mutantes a los que matan (Figura 3).



Figura 3. Estructura de los elementos devueltos por la función *execute* función (línea 2 de la Figura 2)

A la función encargada de recoger los casos que matan n mutantes se le llama en la línea 7 de la Figura 2, y aparece detallada en la Figura 4: recorre los elementos incluidos en *testCaseResults* y toma aquellos casos cuya lista de mutantes muertos (elemento *killedMutants* de la Figura 3) tienen n elementos. Para garantizar que no se seleccionan dos casos que matan a los mismos mutantes, la

función elimina los mutantes muertos del conjunto cada vez que elige un caso de prueba.

```

1. getTestCasesThatKillN(completeTC : SetOfTestCases, n : int, mutants : SetOfMutants, mutantsNowKilled : SetOfMutants, testCaseResults: SetOfTestCaseResults)
2. testCasesThatKillN = ∅
3. for i=1 to |testCaseResults|
4.   testCaseResult = testCaseResults[i]
5.   if |testCaseResult.killedMutants| == n then
6.     testCasesThatKillN = testCasesThatKillN ∪ testCaseResult.testCaseName
7.     mutantsNowKilled = mutantsNowKilled ∪ testCaseResult.killed.Mutants
8.     mutants = mutants - mutantsNowKilled
9.     for j=1 to |testCaseResults|
10.      aux = testCaseResults[j]
11.      if aux.testCaseName ≠ testCaseResult.testCaseName then
12.        aux.remove(mutantsNowKilled[i])
13.      end_if
14.    next
15.  end_if
16. next
17. return testCasesThatKillN
18. end

```

Figura 4. Función que devuelve los casos que matan n mutantes

3.1. Análisis de coste

El coste real de ejecución del algoritmo depende de la clase sobre la que se aplica.

Un caso extremo sucede cuando cada caso de prueba mate solo un mutante (lo cual es prácticamente imposible). En esta situación, la función de la Figura 2 hará n iteraciones (n es el número de mutantes), pero la instrucción condicional de la línea 8 no será cierta hasta que $n=1$. En este caso, el coste computacional del algoritmo es:

$$O(n-1) \cdot O(\text{getTestCasesThatKillN}) + O(1) \cdot O(\text{getTestCasesThatKillN}) + O(|\text{testCases}|)$$

El coste de la función de la Figura 4 hace siempre dos iteraciones (bucles de las líneas 3 y 9): su coste, en el caso peor, será siempre $O(|\text{testCaseResults}|)$.

Ya que $|\text{testCaseResults}| \leq |\text{testCases}|$, el coste del algoritmo en el peor caso es:

$$O(\text{reduceTestSuite}) = O(|\text{testCases}|)^3$$

Otro caso extremo (también casi imposible) sucederá cuando un caso de prueba mate todos los mutantes. En esta situación, los bucles de la Figu-

ra 2 harán una sola iteración, con lo que el coste en este caso es $O(|\text{testCases}|)^2$.

En otras situaciones, el coste computacional depende también del número de mutantes, como se muestra en la Figura 5.

$$O(\text{reduceTestSuite}) = O(|\text{mutants}|) \cdot O(|\text{testCases}|)^3$$

Figura 5. Coste General de *reduceTestSuite*

En la implementación, el coste del algoritmo se reduce respecto del mostrado en la Figura 5 debido al uso de estructuras de datos que permiten búsquedas no secuenciales. Así, el conjunto de objetos de tipo *TestCaseResult* se implementa como una tabla hash indexada por el número de mutantes muertos, lo que hace innecesario recorrer el conjunto completo cada vez que se ejecuta la función de la Figura 4.

4. “A motivational example” (Un ejemplo motivador)

En su artículo [11], Jeffrey y Gupta incluyen una sección con el mismo título de esta (*A motivational example*) en el que muestran un pequeño programa, al que aplican su algoritmo de reducción de casos mediante redundancia selectiva. Para nuestro caso, hemos traducido su código al pequeño programa Java que se muestra en la Figura 6. Para este sencillo programa, MuJava genera 48 mutantes tradicionales (entre los que hay 15 funcionalmente equivalentes) y 6 mutantes de clase.

Usando los valores $\{-1.0, 0.0, -1.0\}$ como valores de prueba para los cuatro parámetros de la función f , y generando los casos de prueba con la estrategia *All combinations* algoritmo, la herramienta *testooj* genera un fichero JUnit y otro MuJava con $3 \times 3 \times 3 \times 3 = 81$ casos de prueba equivalentes, que consiguen matar al 100% de los mutantes no equivalentes.

Tras aplicar el algoritmo de reducción, el *test-suite* de 81 casos se consigue reducir a otro compuesto por solo 7 casos de prueba sin perder cobertura, lo que supone una disminución del tamaño del conjunto al 8.6% del tamaño original.

5. Implementación

La herramienta que implementa los algoritmos mostrados anteriormente requiere un fichero con los casos de prueba en formato MuJava (en la figura, el fichero *MuJavaJGExample_1.class*), la

configuración correspondiente del entorno (variable CLASSPATH), el nombre de la clase bajo prueba (en la figura, *paper.JGExample.class*) y la ubicación de los mutantes de la clase bajo prueba. De acuerdo con el algoritmo de la Figura 7, la herramienta ejecuta la clase con los casos de prueba contra todas las versiones que va encontrando de la clase bajo prueba (lo que incluye, desde luego, a la clase original): en el caso del *JGExample*, se crean 81 instancias de la clase bajo prueba (correspondientes a los 81 casos de prueba) para los 54 mutantes no equivalentes, lo que significa que se crean $81 \times 54 = 4374$ instancias de la clase bajo prueba.

```
public class JGExample {
    float returnValue;
    public float f(float a, float b, float c, float d) {
        float x=0, y=0;
        if (a>0) x=2;
        else x=5;
        if (b>0) y=1+x;
        if (c>0)
            if (d>0)
                returnValue=x;
            else
                returnValue=10;
        else
            returnValue=(1/(y-6));
        return returnValue;
    }
}
```

Figura 6. El “ejemplo motivador” de Jeffrey y Gupta

El resultado de ejecutar cada caso de prueba contra las versiones del programa se coloca en objetos de tipo *ExecutionResult* (cuya estructura se corresponde con la Figura 7), que guardan el estado de la instancia de la clase bajo prueba en forma de cadena junto al nombre del caso de prueba. Con objeto de preservar la memoria del computador, estos resultados de ejecución se guardan en disco, para realizar posteriormente las comparaciones los cálculos mostrados en los algoritmos presentados anteriormente.

Cuando todos los casos han sido ejecutados contra la clase original y los mutantes, la herramienta muestra un resumen de los resultados en una matriz de “mutantes muertos”, cuyas celdas indican que el caso de prueba que se muestra en la columna de la tabla ha matado al mutante de la fila correspondiente. Este resultado se puede

exportar en formatos *html* o *txt* para facilitar otros análisis con, por ejemplo, una hoja de cálculo. Como se ha explicado, la herramienta construye un fichero JUnit con los casos de prueba correspondientes al conjunto reducido.

```
for each cutFile in folders
    cut = load the CUT corresponding to cutFile using
a class loader
    mutantName = name of the folder
    results = ∅
    for each testCase in Suite
        cutInstance = execute testCase on cut
        executionResult = new
            ExecutionResult(testCase.name, cutIn-
stance)
        results = results ∪ {executionResult}
    next
    save results in a file called mutantName.ser
next
```

Figura 7. Algoritmo de ejecución de casos de prueba

6. Validación adicional

Además de al ejemplo ilustrativo de Jeffrey y Gupta, la implementación del algoritmo se ha aplicado a un conjunto de programas que se pueden clasificar en tres categorías:

- *Programas de “juguete”*: se han utilizado los seis programas que Pargas y Harrold [12] utilizan para validar su algoritmo de generación de casos de prueba.
- *Programas industriales*: se ha utilizado la clase *PluginTokenizer* de la aplicación *jtopas*, incluida en la infraestructura de testing de [13] y una clase de tipo contenedor (el *Vector* del paquete *java.util*), también utilizadas en publicaciones sobre testing [14].
- *Programas de estudiantes*: se ha utilizado una clase *Sudoku* escrita por un grupo de alumnos. La clase tiene un constructor sin parámetros que inicializa una matriz de 9×9 enteros; su método *setNumber(int fila, int columna, int numero)* comprueba si es posible poner el *numero* en la *fila* y *columna* según las reglas del sudoku.

Para estos programas, se han generado casos de prueba utilizando *testooj* con *All combinations*. Los resultados se muestran en la Tabla 2. Como se observa, se consiguen en muchos casos reducciones próximas al 90%.

Programa	Mutantes	Nº de casos	
		Original	Reducido
Bisect	66	6	1
Bub	86	84	2
Find	166	252	2
Fourballs	190	64	4
Mid	172	12	2
TriTyp	258	216	22
PluginTokenizer	102	14	1
Vector	444	56	5
Sudoku	135	64	5

Tabla 2. Resultados obtenidos en diversos casos

7. Conclusiones

Este artículo ha presentado un algoritmo de reducción del conjunto de casos de prueba. El algoritmo utiliza el porcentaje de mutantes muertos como criterio de inclusión. El algoritmo se ha implementado como parte de la herramienta *testooj* (<http://alarcos.inf-cr.uclm.es/testooj3>).

De acuerdo con la revisión realizada a la literatura especializada, esta la primera herramienta que incluye un algoritmo de reducción de casos aplicable a herramientas de prueba ampliamente extendidas, como JUnit.

8. Agradecimientos

Trabajo parcialmente financiado por la Dir. Gral. de Investigación/FEDER, TIN2006-15175-C05-05 y TIN2005-24792-E.

9. Referencias

- Polo M, Tendero S, and Piattini M, *Integrating techniques and tools for testing automation*. Software Testing, Verification and Reliability, 2007. **15**(1), 3-39.
- Ng SP, Murnane T, Reed K, Grant D, and Chen TY. *A Preliminary Survey on Software Testing Practices in Australia*. Proceedings of the Australian Software Engineering Conference (ASWEC 2004). 2004. Melbourne, Australia: IEEE Computer Society, pp. 116-125.
- Do H, Rothermel G, and Kinner A, *Prioritizing JUnit test cases: An empirical assessment and cost-benefit analysis*. Empirical Software Engineering, 2005.
- Ma Y-S, Offutt J, and Kwon YR, *MuJava: an automated class mutation system*. Software Testing, Verification and Reliability, 2005. **15**(2), 97-133.
- Polo M and Piattini M. *Automatización del proceso de pruebas unitarias*. PRIS 2006: Taller sobre pruebas en Ingeniería del Software. 2006. Sitges (Barcelona), pp.
- Jones JA and Harrold MJ, *Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage*. IEEE Transactions on Software Engineering, 2003. **29**(3), 195-209.
- Harrold M, Gupta R, and Soffa M, *A methodology for controlling the size of a test suite*. ACM Transactions on Software Engineering and Methodology, 1993. **2**(3), 270-285.
- Heimdahl MPE and George D. *Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing*. 19th IEEE International Conference on Automated Software Engineering (ASE'04). 2004, pp. 176-185.
- McMaster S and Memon AM. *Call Stack Coverage for Test Suite Reduction*. 21st IEEE International Conference on Software Maintenance. 2005. Budapest (Hungary), pp. 539-548.
- Grindal M, Offutt J, and Andler SF, *Combination testing strategies: a survey*. Software Testing, Verification and Reliability, 2005(15), 167-199.
- Jeffrey D and Gupta N. *Test suite reduction with selective redundancy*. International Conference on Software Maintenance. 2005. Budapest (Hungary): IEEE Computer Society Press, pp. 1-10.
- Pargas RP, Harrold MJ, and Peck RR, *Test-Data Generation Using Genetic Algorithms*. Software Testing, Verification and Reliability, 1999(9), 263-282.
- Do H, Elbaum SG, and Rothermel G, *Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact*. Empirical Software Engineering, 2005. **10**(4), 405-435.
- Ball T, Hoffman D, Ruskey F, Webber R, and White L, *State generation and automated class testing*. Software Testing, Verification and Reliability, 2000. **10**, 149-170.