

Generación de mutantes con algoritmos genéticos

Domínguez Jiménez, J. J., Estero Botaro, A., Medina Buló, I.

Departamento de Lenguajes y Sistemas Informáticos, Universidad de Cádiz
C/ Chile, 1, CP 11003, Cádiz
{Juanjose.dominguez, Antonia.estero, immaculada.medina}@uca.es

Resumen. Las técnicas de caja blanca y, más precisamente, las técnicas de prueba de mutaciones dependen del lenguaje de programación empleado. El trabajo que se expone presenta un generador de mutantes independiente del lenguaje de programación empleado, mediante el empleo de algoritmos genéticos. Una de las características del generador es que no se necesitan generar todos los mutantes posibles, sino que la heurística aplicada permitirá realizar una selección de los mejores mutantes. El generador es capaz de detectar los posibles mutantes equivalentes.

Palabras Clave: Prueba de mutaciones, Generador de mutantes, Algoritmos genéticos.

1 Introducción

Las técnicas de caja blanca y, más en concreto, la prueba de mutaciones, tienen un alto grado de dependencia del lenguaje de programación, por lo que no podemos reutilizar herramientas empleadas en otros lenguajes. En este sentido, es necesario el desarrollo de herramientas que, de forma automática, generen mutantes para cualquier lenguaje de programación. Los algoritmos genéticos (AG) han demostrado ser una estrategia de optimización heurística eficaz para optimizar funciones con numerosos óptimos locales. Sin embargo, dentro del ámbito de pruebas del software, su uso se ha limitado a la generación de casos de prueba [8].

Este artículo presenta un generador de mutantes independiente del lenguaje de programación, empleando AG. Nuestro trabajo presenta un uso novedoso de éstos al aplicarlos a la generación de mutantes. El generador será capaz de detectar los posibles mutantes equivalentes. La estructura del artículo es la siguiente: en la sección 2 se resumen brevemente los conceptos teóricos sobre los que subyace el generador; en la sección 3 describimos brevemente la situación actual en la generación automática de mutantes; la sección 4 expone la generación de mutantes propuesta y, finalmente, en la sección 5 se resumen las conclusiones y las futuras líneas de trabajo.

2 Conceptos teóricos

En esta sección se resumen los conceptos principales sobre los que se apoya el generador de mutantes propuesto: la prueba de mutaciones y los AG.

2.1 Prueba de Mutaciones

La prueba de mutaciones es una técnica de prueba del software de caja blanca basada en errores [14], que consiste en introducir fallos simples en el programa original, aplicando para ello *operadores de*

mutación. Los programas resultantes reciben el nombre de *mutantes*. Cada operador de mutación se corresponde con una categoría de error típico que puede cometer el programador. Si un caso de prueba es capaz de distinguir al programa original del mutante, es decir, la salida del mutante y la del programa original son diferentes, se dice que mata al mutante. Si por el contrario ningún caso de prueba es capaz de diferenciar al mutante del programa original, se habla de un mutante vivo para el conjunto de casos de prueba empleado.

Una de las principales dificultades de aplicar la prueba de mutaciones es la existencia de *mutantes equivalentes*. Estos presentan el mismo comportamiento que el programa original. Estos mutantes no deben confundirse con los mutantes difíciles de matar (*stubborn non-equivalent mutants*), originados porque el conjunto de casos de prueba no es suficiente para poder detectarlos.

Uno de los principales inconvenientes de la prueba de mutaciones es el alto coste computacional que implica. Esto es debido a que disponemos normalmente de un número grande de operadores de mutación que generan un elevado número de mutantes. Existen diversas estrategias para reducir el elevado coste computacional de la prueba de mutaciones [11]. Una de ellas es la realización de una *mutación selectiva*, donde se realizan pocas mutaciones sin incurrir en una pérdida de información. Por lo tanto, con objeto de determinar qué mutaciones son las más apropiadas, es interesante la aplicación de técnicas de optimización, como son los AG.

2.2 Algoritmos Genéticos

Los AG [9] son técnicas de búsqueda probabilística basadas en la teoría de la evolución y la selección natural, principalmente en la supervivencia de los mejores y el carácter hereditario de las características. Los puntos fuertes de los AG son su flexibilidad, simplicidad y capacidad de hibridación. Entre sus puntos débiles están su naturaleza heurística y el manejo de restricciones.

Los AG trabajan con una *población* de soluciones, denominadas *individuos*, y procesan toda la información que ésta contiene de forma paralela. A lo largo de las distintas generaciones de la población, los AG realizan un proceso de selección y mejora de los individuos, de manera que son ideales para la resolución de problemas de optimización.

El esquema de codificación empleado depende de las características del problema que se trate. Cada individuo tendrá una aptitud que medirá la calidad de la solución que representa respecto al problema que se está resolviendo. La aptitud de los individuos será objeto de un proceso de maximización a lo largo de las distintas generaciones del algoritmo.

En un AG existen dos tipos de operadores: de selección y de reproducción. Los *operadores de selección* se encargan de seleccionar individuos de una población para la reproducción. Los *operadores de reproducción* permiten la generación de nuevos individuos en la población. Podemos distinguir dos tipos: el cruce y la mutación. El *operador de cruce* genera dos individuos nuevos, denominados *hijos*, a partir de dos individuos seleccionados previamente, denominados *padres*. Los hijos heredan parte de la información almacenada en cada uno de los dos padres. El *operador de mutación* tiene como finalidad alterar la información almacenada en un individuo. El diseño de estos operadores depende del esquema de codificación empleado.

3 Antecedentes

La generación automática de mutantes ha permitido evaluar la efectividad de las diferentes técnicas de prueba del software, y así mejorar su eficiencia. Existen diversos trabajos en la bibliografía que tratan sobre el desarrollo de herramientas para automatizar la generación de mutantes, tales como Mothra [5] para Fortran, MuJava [6] para Java, Proteum [3] para C, Insure [12] para C++, SQLMutation [13] para SQL. Todos estos generadores se caracterizan porque producen de forma automática todos los mutantes posibles para los operadores de mutación proporcionados, sin realizar ninguna mutación selectiva en la generación.

Existen en la bibliografía numerosas aplicaciones de los AG a las pruebas del software [7], aunque la mayor parte de éstas se han limitado a la generación de casos de prueba. En cuanto a la aplicación de AG a la generación de mutantes, existen pocas referencias. Así, en la búsqueda bibliográfica realizada, sólo Adamopoulos y col. [1] describe un sistema para la generación de mutantes y casos de prueba mediante AG en combinación con Mothra. En ese sistema, los individuos del AG están formados por conjuntos de mutantes generados previamente con Mothra, a diferencia de nuestro trabajo, donde los mutantes se generan con el propio AG.

4 Generación de Mutantes con Algoritmos Genéticos

El generador de mutantes que se presenta está basado en AG. Además de la generación automática de mutantes, se pretende resolver la detección de los mutantes equivalentes al programa original. La figura 1 muestra el esquema del generador. Podemos distinguir dos elementos: un analizador del programa original y el generador de mutantes. El sistema tiene como entrada un programa original, un conjunto inicial de casos de prueba, así como los parámetros de configuración del AG.

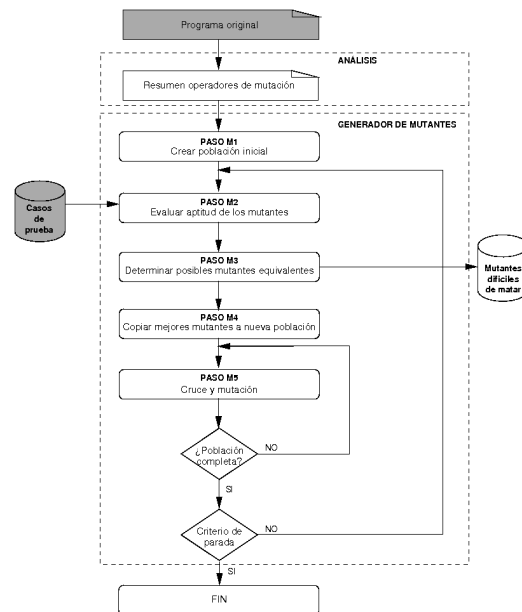


Fig. 3. Esquema del generador de mutantes

Antes de ejecutar el generador de mutantes, el sistema analizará el programa original, de manera que se identificarán las instrucciones que pueden sufrir mutación de acuerdo a los operadores de mutación definidos para el lenguaje. El componente analizador será el único que dependa del lenguaje. A continuación se ejecutará el generador de mutantes, cuyo objetivo es generar de forma automática mediante un AG los mutantes del programa original.

4.1 Analizador

Esta fase consiste en la identificación de las distintas instrucciones o elementos del programa original que pueden sufrir modificación de acuerdo al conjunto de operadores de mutación definidos para el lenguaje. De esta manera, al final obtendremos un fichero de texto que contiene para cada operador de mutación un recuento de las instrucciones a las que se puede aplicar.

4.2 Generador de Mutantes

La generación de mutantes se realiza mediante un AG, en el que cada individuo representa a un mutante. A continuación pasamos a detallar los distintos aspectos del algoritmo.

Representación de los Individuos. Cada individuo codifica la mutación a realizar al programa original mediante tres campos: uno para identificar el operador, otro para referenciar la instrucción donde se aplicará, y un tercero que contiene información para la aplicación del operador.

El campo *Operador* identifica al operador de mutación que se aplica. Se codifica con un valor entero entre 1 y OM, donde OM es el número de operadores de mutación definidos.

El campo *Instrucción* representa el número de instrucción del programa original donde se aplicará el operador. Con objeto de realizar una distribución uniforme entre todos los operadores independientemente del número de instrucciones asociadas a cada uno, el campo se codifica con un valor entero comprendido en el rango de 1 a I, donde $I = mcm \{m_i, 1 \leq i \leq OM\}$, siendo m_i el número de instrucciones que existen en el programa original a las que se puede aplicar el operador de mutación i -ésimo. Posteriormente, es necesario realizar una traducción, de manera que el valor I_i representará una operación de mutación en la instrucción $\lceil (I_i \times m_i) / I \rceil$

El campo *Atributo* representa la información necesaria para la aplicación del operador de mutación. Dado que una mutación consiste en la alteración de un elemento del programa, este campo especifica cuál será el nuevo valor que tomará el elemento afectado por la mutación. Así, los operadores de mutación podemos clasificarlos en:

- Aquellos que cambian un elemento por otro de un conjunto. En este caso, el atributo indica el nuevo elemento que se quiere poner. Consideremos, por ejemplo, los operadores de mutación para las expresiones relacionales que aparecen en la tabla 1 con sus respectivos valores. El individuo que indicaría cambiar la instrucción $a > 2500$ por $a < 2500$, se codificaría con un 3.

Tabla 1. Operadores relacionales y su correspondiente valor

| Operador | > | = | < | >= | <= | != |
|----------|---|---|---|----|----|----|
| Valor | 1 | 2 | 3 | 4 | 5 | 6 |

- Aquellos que eliminan un elemento. En este caso, el atributo no tendría significado. Se mantiene en la codificación con valor 1. Un ejemplo de este tipo de operadores pueden ser aquellos que eliminan una rama `else` de una construcción `if-else`.
- Aquellos que suponen un cambio de posición de la instrucción. El atributo indicará cuál es la nueva posición donde se ubicará. Si se intercambia con la instrucción anterior se codifica con 1, y si es con la instrucción posterior se codifica con 2.

Al igual que en el campo *Instrucción*, con objeto de realizar una distribución uniforme entre todos los individuos, el campo *Atributo* contiene un valor entero dentro del rango 1 a V, donde $V = mcm \{v_i, 1 \leq i \leq OM\}$, siendo v_i el número de valores que puede tomar el operador de mutación i -ésimo. De forma análoga se realiza una traducción para la obtención del desplazamiento.

Este esquema de codificación de individuos permite que la arquitectura pueda adaptarse fácilmente a cualquier lenguaje de programación. Simplemente habría que numerar los distintos operadores de mutación disponibles y, para cada uno de ellos, sus posibles valores.

Aptitud de los Individuos. Para poder evaluar la aptitud de un individuo (fig.1, paso M2) necesitamos ejecutar el mutante al que representa frente a los casos de prueba hasta que encontremos uno que lo mate. Al finalizar la ejecución de todos los mutantes, dispondremos de una matriz de dimensiones $M \times T$, siendo M el número máximo de mutantes y T el número máximo de casos de prueba de que disponemos, donde cada elemento m_{ij} de la matriz valdrá 1 ó 0, dependiendo de si el mutante i -ésimo ha sido matado o no con el caso de prueba j -ésimo. En dicha matriz, se cumplen las siguientes relaciones:

- Un mutante puede ser matado por un caso de prueba o seguir vivo: $\sum_{j=1}^T m_{ij} \in [0,1] \forall i$
- Un caso de prueba puede matar a ninguno o a varios mutantes: $\sum_{i=1}^M m_{ij} \in [0, M], \forall j$

Cuando un caso de prueba no mata a ningún mutante puede indicarnos dos cosas: el caso de prueba tiene baja calidad, o bien los mutantes generados no son adecuados para ese caso de prueba.

La función de aptitud representa al número de mutantes que son matados con el mismo caso de prueba. Dado que el algoritmo genético requiere siempre una función a maximizar, se establece una cota superior a dicho valor. Así, la función de aptitud para un individuo I :

$$Aptitud(I) = M - \sum_{j=1}^T \left(m_{ij} \cdot \sum_{i=1}^M m_{ij} \right) \quad (2)$$

Esta función de aptitud de un individuo tendrá en cuenta si éste es muerto o no por los casos de prueba y cuántos mutantes son también matados por el mismo caso de prueba. De este modo, se intenta penalizar aquellos grupos de mutantes que son matados por el mismo caso de prueba, y favorecer a aquellos que son matados por un caso de prueba que sólo mata a dicho mutante, y por tanto, difíciles de matar. Sin embargo, cuando un mutante no es matado por ningún caso de prueba, es decir, no es detectado (tiene una aptitud de valor M), será almacenado en una memoria del algoritmo (fig. 1, paso M3), considerándose como un posible mutante equivalente.

Operadores. La primera población del AG será generada aleatoriamente (fig. 1, paso M1). El tamaño de la población constituye un parámetro de entrada del algoritmo. Se emplea un AG generacional, donde los individuos de las siguientes generaciones provienen tanto de la población anterior, como de las operaciones de reproducción. Así, el algoritmo tendrá un parámetro de entrada, porcentaje de repetidores, que determina cuántos mutantes de la generación antigua pasan a la nueva. La selección de estos mutantes se realiza mediante elitismo, escogiendo aquellos individuos con mejor aptitud (fig. 1, paso M4).

Por otro lado, el algoritmo genera nuevos individuos mediante la realización de operaciones de cruce y mutación sobre los individuos de la población anterior. La selección de los individuos participantes se realiza mediante la técnica de la ruleta [4]. Se realizan operaciones de cruce o mutación, ambas excluyentes, de acuerdo a las probabilidades de estas operaciones, que mantendrán la siguiente regla: $p_m = 1 - p_c$, donde p_m es la probabilidad de realizar una operación de mutación y p_c la probabilidad de cruce, siendo éste un parámetro de entrada del algoritmo.

Operadores Genéticos. El AG para la generación de nuevos individuos (fig. 1, paso M5) aplica los dos operadores genéticos: el cruce y la mutación. La selección de los individuos participantes se realiza con probabilidad uniforme, sin tener en cuenta la aptitud de los mismos.

El operador de cruce consiste en realizar un cruce básico de un punto. El objetivo de éste es realizar un intercambio de elementos de los individuos participantes, de manera que el punto de cruce se escoge aleatoriamente entre los posibles campos que componen los individuos.

El operador de mutación consiste en cambiar el valor de un elemento del individuo. De este modo existen 3 tipos de operadores de mutación, según donde se apliquen: mutación de instrucción, mutación de operador y mutación de atributo.

Dado que las tres mutaciones cambian elementos codificados como enteros, se empleará una mutación consistente en generar una perturbación gaussiana aleatoria alrededor de uno de los valores del individuo seleccionado. La magnitud de la perturbación que se realiza se controla a través de la desviación estándar θ cuyo valor se mantiene constante en todo el proceso de funcionamiento del algoritmo. Esta magnitud está relacionada con la probabilidad de mutación p_m , de forma que si las mutaciones son muy frecuentes, el tamaño de la magnitud de éstas decrece, y viceversa, es decir, $\theta = 1 - p_m$.

Criterio de Parada. El AG finalizará cuando se alcance un número máximo de generaciones.

5 Conclusiones y Trabajo Futuro

Este artículo presenta un generador de mutantes basado en AG para la realización de prueba de mutaciones, independiente del lenguaje de programación. La principal aportación es una propuesta heurística para la generación heurística de mutantes, sin necesidad de generar todos los mutantes

posibles, reduciendo el alto coste computacional de la prueba de mutaciones. Otra característica importante del generador propuesto es su independencia del lenguaje de programación para el que se crean los mutantes. Para ello, los individuos del AG codifican la instrucción a mutar y el operador de mutación que se empleará. De este modo, es fácilmente adaptable a cualquier otro lenguaje de programación.

El generador tiene además como objetivo secundario la detección de posibles mutantes equivalentes. Para ello, el generador de mutantes dispone de una memoria donde almacena aquellos que son difíciles de matar.

Nuestro trabajo se centra ahora en la realización de las primeras pruebas con el generador. Hemos escogido el lenguaje WS-BPEL 2.0 [10] al no disponer actualmente de generadores automáticos de mutantes. Para ello estamos elaborando un conjunto de operadores de mutación, e identificando los atributos que tiene cada operador. Una vez realizada esta labor, podremos realizar las primeras pruebas de su funcionamiento con composiciones de servicios web en WS-BPEL [2]. Posteriormente, podremos comparar la efectividad de la herramienta empleándola en lenguajes que ya disponen de otros generadores de mutación.

El generador presentado se convierte en el primer paso del desarrollo de una herramienta para la prueba de mutaciones donde, conectado con un generador de casos de prueba, permita generar casos de prueba de calidad. Por otro lado, el generador de mutantes proporcionará información de la evolución a lo largo de las distintas generaciones. De este modo, podremos analizar qué operadores de mutación son más efectivos, y así determinar un subconjunto de operadores óptimo.

Agradecimientos

Este trabajo ha sido financiado por el Programa Nacional de I+D+I del Ministerio de Educación y Ciencia y fondos FEDER mediante el proyecto SOAQSIm (TIN2007-67843-C06-04).

Referencias

1. Adamopoulos, K., Harman, M., Hierons, R.M.: How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. Proc. of the GECCO'04, pp. 1338–1349 (2004)
2. Canfora, G., Di Penta, M.: Testing services and service-centric systems: challenges and opportunities. IT Professional, vol. 8(2), pp. 10–17 (2006)
3. Delamaro, M., Maldonado, J.: Proteum—A Tool for the Assessment of Test Adequacy for C Programs. Proc. of the Conference on Performability in Computing System, pp. 79–95 (1996)
4. Goldberg, D.E.: Genetic algorithms in search, optimization and machine learning. Addison-Wesley (1989)
5. King, K.N., Offutt, A.J.: A Fortran Language System for Mutation-based Software Testing. Software - Practice and Experience, vol. 21(7), pp. 685–718 (1991)
6. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: an automated class mutation system. Soft. Test., Verif. & Rel., vol. 15(2), pp. 97–133 (2005)
7. Mantere, T., Alaner, J.T.: Evolutionary Software Engineering, a review. Applied Soft Computing, vol. 5, pp. 315–331 (2005)
8. Mcminn, P.: Search-based software test data generation: A survey. Soft. Test., Verif. & Rel., vol. 14(2), pp. 105–156 (2004)
9. Michalewicz, Z.: Genetic algorithms + data structures = Evolution programs. Springer-Verlag (1992)
10. OASIS: WS-BPEL 2.0 (2007), <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>
11. Offutt, A.J., Untch, R.H.: Mutation 2000: uniting the orthogonal. Mutation testing for the new century, pp. 34–44, Kluwer Academic Publishers, Norwell, MA, USA (2001)
12. Parasoft: Insure++ (2008), <http://www.parasoft.com/jsp/products/home.jsp?product=Insure>
13. Tuya, J., Suárez-Cabal, M.J., de la Riva, C., SQLMutation: a Tool to Generate Mutants of SQL Database Queries. Proc. of the Second Workshop on Mutation Analysis, Raleigh, (2006).
14. Woodward, M.R.: Mutation testing —its origin and evolution. Information and Software Technology, vol. 35(3), pp. 163–169 (1993)