Ingeniería Industrial/ISSN 1815-5936/Vol. XXXVI/No. 2/mayo-agosto/2016/p. 200-207

ARTÍCULO ORIGINAL
GESTIÓN DEL CONOCIMIENTO

Generación de combinaciones de valores de pruebas utilizando metaheurísticas

Generation of combinations of test values using metaheuristics

Perla Fernández-Oliva[I], William Cantillo-Terrero[I], Martha Dunia Delgado-Dapena[I], Alejandro Rosete-Suárez[I], Cornelio Yáñez-Márquez[II]

[I] Instituto Superior Politécnico José Antonio Echeverría, CUJAE. Facultad de Ingeniería Informática. La Habana, Cuba.
E-mail: perla@ceis.cujae.edu.cu, wcantillo@ceis.cujae.edu.cu, marta@ceis.cujae.edu.cu, rosete@ceis.cujae.edu.cu

[II] Centro de Investigación en Computación del Instituto Politécnico Nacional, México, D.F. México.
E-mail: coryanez@gmail.com

Recibido: 2/02/2016
Aprobado: 27/04/2016

RESUMEN
La fase de prueba es un proceso difícil que consume un porcentaje elevado del costo en cuanto al tiempo del proceso de desarrollo del software. La finalidad de las pruebas es determinar si los productos desarrollados cumplen los requisitos acordados con los usuarios y clientes en las especificaciones. Por ello se necesitan los procesos, métodos y herramientas que permitan obtener buenos conjuntos de pruebas de un sistema. En este artículo se presenta un componente que combina automáticamente valores para realizar pruebas unitarias y para eso se aplican algoritmos metaheurísticos. La solución propuesta ha sido probada en un caso de estudio y comparada con los valores obtenidos por otros algoritmos propuestos por autores que trabajan el tema en la comunidad científica. El componente permite obtener un conjunto reducido de valores de prueba, en un tiempo de ejecución menor y con una cobertura del 100%.
Palabras clave: Combinación automática de valores, algoritmos de optimización, algoritmos metaheurísticos, pruebas unitarias.

ABSTRACT
The test phase is a difficult process that consumes a large percentage of the cost in terms of the time of the process of software development. The purpose of the tests is to determine if the developed products meet the requirements accord with the users and customers on specifications. Therefore, the processes, methods and tools that allow obtain good sets of test of a system are needed. This article introduces a component that automatically combines values to perform unit tests and for that were applied metaheuristics algorithms. The proposed solution has been tested in a case study, and was compared with values obtained through other algorithms proposed by authors who work the issue in the scientific community. The component allows to obtain a reduced set of test values, with less time of execution and a coverage of 100%.

Key words: Automatic combination of values, optimization algorithms, metaheuristics algorithms, unit tests.

I. INTRODUCCIÓN

Según Myers las pruebas son muy costosas por lo que se dejan para las últimas etapas del proyecto y no se realizan con la calidad necesaria [1]. No obstante, existen múltiples propuestas que se centran en la planificación y cálculo de los medios indispensables para realizarlas [2; 3; 4]. Así como a la generación automática de escenarios, y valores de prueba[1][5; 6; 7]. La generación automática de un conjunto de datos de prueba que permita detectar los errores de un programa dado en la fase de prueba, es una tarea de suma importancia que requiere un alto costo computacional y que generalmente es difícil de resolver [8; 9].

Harmany McMinn exploran respuestas que brindan las temáticas de la Ingeniería de Software **Basada en Búsquedas** para dar solución a problemas combinatorios utilizando métodos de optimización [10; 11; 12; 13]. Existen trabajos recientes que automatizan la realización de las pruebas de software con respecto a la generación de escenarios y valores de prueba, utilizando técnicas para evadir la explosión combinatoria [10; 14; 15].

Ribeiro y Ahmed describen el empleo de algoritmos de búsqueda para la generación de casos de prueba para programas orientados a objetos desarrollados en Java [14; 16]. Varshney hace un recorrido por las diversas técnicas de búsqueda que se han aplicado para la generación de datos de prueba estructural[2,3] [17; 18; 19; 20]. Díaz propone un modelo puro basado en el algoritmo **Búsqueda Tabú** para la generación automática de valores para casos de prueba [21]. Mientras que en se presenta una solución basada en la fusión de una metaheurística poblacional con una lista Tabú [22].

Las propuestas existentes de generación de valores no tienen en cuenta el hecho de que una combinación de valores puede abarcar un camino de prueba representado por una funcionalidad [17]. Por lo cual este elemento podría reducir el número de combinaciones de valores, de forma tal que se reduzca el número total de combinaciones en función del camino que se quiere probar. Es necesario reducir las combinaciones de los valores de prueba, integrando la lógica interna del programa con las combinaciones de los valores de cada variable.

El objetivo de este artículo es presentar un componente con una nueva solución al problema de generar el conjunto de valores adecuados para la prueba al software basado en el Escalador de Colina.

El componente de generación de valores de prueba utilizará la información de la estructura del programa teniendo en cuenta los caminos independientes de pruebas y el listado de valores significativos para cada variable para así guiar la generación de los valores.

II. MÉTODOS

Componente para la generación de combinaciones de valores de pruebas

A continuación se definen elementos más importantes para presentar la propuesta del componente.

El objetivo fundamental del componente desarrollado es devolver un conjunto reducido de combinaciones de valores que cubran una serie de caminos de pruebas para realizar pruebas unitarias. El criterio de cobertura empleado en la solución es el de los caminos. Este componente estará utilizando la información necesaria de tres ficheros de entradas de datos:

1- las variables del problema y el dominio que pueden tomar las mismas para un caso de prueba en específico.

---

[1] Mejías, L.N.M., *Componente de generación automática de valores de pruebas.*, in *Facultad de Ingeniería Informática*. 2012, Instituto Superior Politécnico José Antonio Echeverría. p. 72.

[2] Ferguson, R. and B. Korel, *The chaining approach for software test data generation.* ACM Transactions on Software Engineering and Methodology (TOSEM), 1996. **5**(1): p. 63-86.

[3] Jones, B.F., H.-H. Sthamer, and D.E. Eyres, *Automatic structural testing using genetic algorithms.* Software Engineering Journal, 1996.11(5): p. 299-306.

2- las condiciones que deben cumplir las variables del problema
3- los casos de pruebas en específico, que no son más que los caminos de pruebas que se desean cubrir.

El componente permite integrar estos tres ficheros ofreciendo al equipo de pruebas un conjunto reducido de combinaciones de valores. De esta forma, este problema perteneciente al área de la Ingeniería de Software es transformado en un problema de optimización ya que se busca reducir el conjunto de valores de prueba.

Criterio de cobertura

El criterio utilizado en este artículo es el de cobertura de caminos. Esto implica que todos los caminos de prueba deben tomar los valores de verdad posibles, es decir, que se busca pasar por todos las ramas del programa. Al exigir que todas las condiciones alcancen ambos valores de verdad, garantiza que todas las ramas serán cubiertas y por lo tanto, todas las instrucciones del programa serán ejecutadas. Para llevar a cabo este criterio, cada uno de los caminos del programa es analizado de manera independiente. Este proceso se lleva a cabo en la evaluación de la Función Objetivo, que se explicará más adelante.

Codificación del problema

Para esta propuesta los valores de las variables representan el dominio que pueden tomar las mismas para probar una funcionalidad determinada. Las condiciones empleadas están formadas por las variables del problema y se obtienen de un grafo de control de flujo el cual representa trayectorias, las cuales determinan los caminos.

De cada variable empleada se obtendrá un valor y se formará un estado. Un estado no es más que la combinación de la asignación de un valor de cada variable.

La solución propuesta en este artículo para este problema solo se aplica a variables de entrada de los tipos de datos numéricos, booleanos, cadenas de caracteres y fechas.

En la tabla 1 se muestra un ejemplo de cómo es asignado un valor a cada variable para un problema con 9 variables de entrada, 7 numéricas y 2 booleanas.

Tabla 1. Variables y valores de un problema

A	B	X	P	I	Fin	F	Xi	R
0	0	0	0	0	0	True	0	True
B	A	P	X	250	250	False	250	False
250	(0+A)2	250	250	Fin	I		P	
(0+B)/2	250	(P+250)/2	(0+X)/2	(0+Fin)/2	(0+Fin)/2		(0+I)/2	

Un estado, como se mencionó antes, se forma mediante la selección de un valor de cada variable respetando el orden que se le dio en la entrada. Esto se muestra en la tabla 2.

Tabla 2. Estado formado

0	A	P	0	250	I	True	250	False

Optimización mediante la biblioteca BiCIAM

La biblioteca de clases BiCIAM que implementa algoritmos metaheurísticos basada en un modelo unificado de estos algoritmos [23; 24]. Esta biblioteca proporciona un modelo que permite dar una solución al problema existente mediante el uso de las metaheurísticas.

Componente de Generación

El componente para la generación de valores de prueba elegido es del tipo de caja blanca, por lo tanto es necesario conocer el valor de las variables involucradas en cada condición. Esto se logra con los ficheros de entrada mencionados anteriormente.

Toda la información generada es evaluada de forma automática por el componente en la función objetivo.

Generación del estado inicial

La construcción aleatoria del estado inicial devuelve una cadena con longitud igual que la cantidad de variables del problema.

Para realizar la construcción de la solución inicial cumpliendo con la codificación propuesta, se ha diseñado la clase M1P la cual es la encargada de ofrecer la solución inicial del problema y generar la vecindad del mismo. Estas clases heredan de la clase abstracta *Operator* que proporciona (BiCIAM). En ella se implementa el método *generate Random State*() el cual se encarga de generar el estado inicial y ofrecer el punto de partida al problema.

A este se le pasa por parámetro la cantidad de estados iniciales que se quiere generar. En este problema se requiere de un solo estado inicial para que los algoritmos tengan un punto de partida.

Partiendo de la cantidad de variables del problema se ejecuta un ciclo, donde se verifica si la variable en la posición i trabaja con valores numéricos. Si la variable en la posición i trabaja con valores numéricos se guarda el nombre de la variable y el primer valor de la misma en un diccionario, *HashMap*.

La clase *HashMap* es un contenedor de almacenamiento de objetos que utiliza Java.

En la segunda parte del método se genera concretamente el estado inicial para el problema. Se obtiene por cada variable un valor aleatorio formando un estado (como se mostró en la sección anterior).

Clase Parser

Parser: Esta clase va a ser invocada primeramente desde la clase que implementa el operador de mutación en un punto y la que genera el estado inicial. En ambos métodos de la clase M1P es invocado el método *parse Numeric Expression*() de la clase *Parser*, al cual se le pasa por parámetro la expresión (que puede ser un número, una variable o combinaciones de variable-variable, variable-número unidos por operadores aritméticos), y un diccionario el cual tendrá los valores actuales de cada variable. Una vez utilizado el método de la clase *Parser* se devolverá un valor numérico para la expresión.

En la Función Objetivo (FO) implementada también se hace uso de la clase *Parser*, en este caso para la lectura e interpretación de las condiciones del problema. Para este caso las condiciones son leídas de forma independiente ofreciendo un valor booleano como respuesta a la evaluación.

La clase Parser para la lectura e interpretación de las expresiones utiliza algunos métodos encargados de identificar los operandos y los operadores existentes en cada expresión.

Los elementos que la clase identifica se muestran en la tabla 3.

Tabla 3. Elementos interpretados

Operadores	Símbolo	Descripción
Relacionales	= =	igual
	! =	distinto
	<	menor que
	>	mayor que
	< =	menor e igual que
	> =	mayor e igual que
Lógicos	&&	Y
	||	O
Aritméticos	+	Suma
	-	Resta
	*	Multiplicación
	/	División
Otros	()	Para agrupar un conjunto de elementos.
	“”	Lo que muestra se interpreta como una cadena de caracteres
	@	indica que luego aparece una fecha

Las expresiones pueden estar relacionadas con cualquiera de los elementos antes mostrados.

Función Objetivo

La función objetivo se encarga de realizar el proceso de evaluación a los diferentes estados que son obtenidos durante la experimentación, es decir, es la encargada de evaluar las restricciones y dar una respuesta.

Por la FO pasan todos los estados generados para ser evaluados y determinar cuáles de ellos cubren los caminos de pruebas deseados. Por lo que se guardan los estados que cubren los caminos de pruebas.

Con las tres primeras estructuras la FO implementada puede evaluar cada estado generado y ofrecer un valor entre 0 y 1 como respuesta de la evaluación. Mientras más cerca esté el valor obtenido de 1, pues entonces habrá posibilidad de que el estado generado este cercano a cubrir un camino.

El resultado de evaluar cada condición forma un camino de valores booleanos al igual que los valores de entrada del fichero de caminos, luego este es comparado con los caminos que se quieren cubrir para determinar si el estado generado y ya evaluado satisface alguno de los caminos de prueba.

Un camino es cubierto si la cantidad de evaluaciones que se cumplieron es igual a la cantidad de evaluaciones que se realizaron por cada camino de prueba. Si esto ocurre, el estado evaluado es guardado.

Del proceso de selección del mejor estado se encargará el algoritmo metaheurístico empleado, el cual escogerá el mayor valor encontrado por cada vecindad generada y evaluada en la función objetivo.

Estructura del fichero de salida

Una vez que se carguen los tres ficheros de entrada y sea ejecutado el componente, el mismo dará la opción de guardar la salida en un fichero texto en el que se mostrarán los estados que cumplieron con los distintos caminos de pruebas a cubrir.

II. RESULTADOS

A continuación se exponen los resultados de la solución desarrollada utilizando un caso de estudio como mecanismo de validación de la propuesta implementada.

La solución se implementó en lenguaje Java. El desempeño del componente propuesto fue probado en la generación de valores de prueba para un método utilizado por diferentes autores.

- El problema de clasificación del triángulo tiene tres variables de entrada (A, B, C) que representan la longitud de los lados de la figura. El programa determina si representa un triángulo, y en ese caso su tipo (escaleno, equilátero, isósceles).

La tabla 4 muestra los resultados obtenidos luego de realizar 2000 iteraciones y el tiempo promedio que tardó en lograr cada propuesta un 100% de cobertura. También se incluyen los resultados al aplicar varios operadores sobre el componente propuesto, como Mutación en un punto (M1P), mutación en 2 puntos (M2P) y mutación en 3 puntos (M3P), bajo las mismas condiciones.

Tabla 4. Comparación de los resultados

Algoritmos propuesto por	Cantidad de combinaciones	Tiempo promedio (segundos)
Jones [3]	17789	8,4
Díaz [4]	587	1,09
Lanzarini [5]	51	1,03
Componente (con M1P)	5	0,853
Componente (con M2P)	5	0,150
Componente (con M3P)	5	0,397

IV. DISCUSIÓN

Del problema de clasificación de triángulo se identificaron 5 caminos de prueba. La solución propuesta en este artículo permite cubrir el 100% de los caminos de prueba, y genera un conjunto reducido de valores para esos caminos. Además, obtiene los resultados en menos tiempo que las propuestas de Jones, Lanzarini y Díaz[4] [22; 23].

El tiempo empleado en el método propio con los diferentes tipos de mutación es menor que los de los autores referenciados.

Se puede observar que de los tres operadores el de mejor comportamiento fue el de M2P. El operador implementado en la clase M1P solo cambia un valor de una variable, provocando que la cantidad de ejecuciones sea mayor. Ello se debe a que, el estado solo cambia en una variable, lo cual lo hace muy simple. El operador implementado en la clase M3P hace que el estado cambie en tres variables, este proceso hace que se transforme mucho un estado generado, lo cual provoca que el proceso de la formación de la vecindad sea más lento.

[4]Jones, B.F., H.-H. Sthamer, and D.E. Eyres, *Automatic structural testing using genetic algorithms.* Software Engineering Journal, 1996.11(5): p. 299-306.

V. CONCLUSIONES

1. Del problema de clasificación de triángulo se identificaron 5 caminos de prueba. La solución propuesta en este artículo permite cubrir el 100 % de los caminos de prueba, y genera un conjunto reducido de valores para esos caminos. Además, obtiene los resultados en menos tiempo que las propuestas de Jones, Lanzarini y Díaz[5] [22; 23].
2. El tiempo empleado en el método propio con los diferentes tipos de mutación es menor que los de los autores referenciados.
3. Se puede observar que de los tres operadores el de mejor comportamiento fue el de M2P. El operador implementado en la clase M1P solo cambia un valor de una variable, provocando que la cantidad de ejecuciones sea mayor. Ello se debe a que, el estado solo cambia en una variable, lo cual lo hace muy simple. El operador implementado en la clase M3P hace que el estado cambie en tres variables, este proceso hace que se transforme mucho un estado generado, lo cual provoca que el proceso de la formación de la vecindad sea más lento.

VI. REFERENCIAS

Myers GJ, Sandler C, Badgett T. The art of software testing. New York: John Wiley & Sons; 2011. ISBN 1118133153.

2. Lamancha BP, Polo M. Generación automática de casos de prueba para Líneas de Producto de Software. Innovación, Calidad e Ingeniería del Software. 2009;5(2):17. ISSN 1885-4486.

3. Memon AM, Pollack ME, Soffa ML. Hierarchical GUI test case generation using automated planning. Software Engineering, IEEE Transactions on. 2001;27(2):144-55. ISSN 0098-5589.

4. Zhang Z, Yan J, Zhao Y, et al. Generating combinatorial test suite using combinatorial optimization. Journal of Systems and Software. 2014;98:191-207. ISSN 0164-1212.

5. Bouquet F, Grandpierre C, Legeard B, et al. A test generation solution to automate software testing. En: Proceedings of the 3rd international workshop on Automation of software test. Leipzig (Germany): ACM; 2011. 45-8. ISBN 1605580309.

6. Lam SSB, Raju MHP, Ch S, et al. Automated generation of independent paths and test suite optimization using artificial bee colony. Procedia Engineering. 2012;30:191-200. ISSN 1877-7058.

7. Anand S, Burke EK, Chen TY, et al. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software. 2013;86(8):1978-2001. ISSN 0164-1212.

8. Jacobson I, Booch G, Rumbaugh J, et al. The unified software development process. Boston: Addison-Wesley Reading; 1999. ISBN 978-0321822000.

9. Harman M, Mansouri SA, Zhang Y. Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR). 2012;45(1):11. ISSN 0360-0300.

10. Harman M. Automated test data generation using search based software engineering. En: Automation of Software Test, 2007 AST'07 Second International Workshop on. IEEE. p. 2-. ISBN 0-7695-2971-2.

11. Harman M, Lakhotia K, Singer J, et al. Cloud engineering is search based software engineering too. Journal of Systems and Software. 2013;86(9):2225-41. ISSN 0164-1212.

12. Iqbal MZ, Arcuri A, Briand L. Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. En: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM. p. 199-209. ISBN 1450314546.

13. Ribeiro JCB. Search-based test case generation for object-oriented java software using strongly-typed genetic programming. En: Proceedings of the 10th annual conference companion on Genetic and evolutionary computation. ACM. p. 1819-22. ISBN 1605581313.

14. Ahmed BS, Zamli KZ. Comparison of metahuristic test generation strategies based on interaction elements coverage criterion. En: Industrial Electronics and Applications (ISIEA). Langkawi (Malasia): IEEE; 2011. p. 550-4. ISBN 1457714183.

[5]Jones, B.F., H.-H. Sthamer, and D.E. Eyres, *Automatic structural testing using genetic algorithms.* Software Engineering Journal, 1996.11(5): p. 299-306.

15. Varshney S, Mehrotra M. Search based software test data generation for structural testing: a perspective. ACM SIGSOFT Software Engineering Notes. 2013;38(4):1-6. ISSN 0163-5948.
16. Ferguson R, Korel B. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology (TOSEM). 1996;5(1):63-86. ISSN 1049-331X.
17. Jones BF, Sthamer H-H, Eyres DE. Automatic structural testing using genetic algorithms. Software Engineering Journal. 1996;11(5):299-306. ISSN 0268-6961.
18. Michael CC, McGraw G, Schatz MA. Generating software test data by evolution. Software Engineering, IEEE Transactions on. 2001;27(12):1085-110. ISSN 0098-5589.
19. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. Information and Software Technology. 2001;43(14):841-54. ISSN 0950-5849.
20. Díaz E, Tuya J, Blanco R, et al. A tabu search algorithm for structural software testing. Computers & Operations Research. 2008;35(10):3052-72. ISSN 0305-0548.
21. Lanzarini LC, Battaiotto PE. Dynamic generation of test cases with metaheuristics. Journal of Computer Science & Technology. 2010;(10). ISSN 1000-9000.
22. Infante AL, André M, Suárez AR, et al. Conformación de equipos de proyectos de software aplicando algoritmos metaheurísticos de trayectoria multiobjetivo. Inteligencia artificial: Revista Iberoamericana de Inteligencia Artificial. 2014;17(54):1-16. ISSN 1137-3601.
23. Abreu ALI, Hernández RD, Ampuero MA, et al. Solución al problema de conformación de equipos de proyectos de software utilizando la biblioteca de clases BICIAM. Revista Cubana de Ciencias Informáticas. 2015;9:126-40. ISSN 2227-1899.
24. Pressman RS. Software engineering: a practitioner's approach. London: Palgrave Macmillan; 2005. ISBN 007301933X.