Vol. VII, No. 2, mayo - agosto, 2016, pp. 48 - 54

Generación automática de combinaciones de valores para pruebas funcionales utilizando metaheurísticas

Artículo Original

Arloys Macías Rojas
correo electrónico: amacias@ceis.cujae.edu.cu
Universidad Tecnológica de La Habana José Antonio Echeverría, Cujae, La Habana, Cuba

Martha Dunia Delgado Dapena
correo electrónico: marta@ceis.cujae.edu.cu
Universidad Tecnológica de La Habana José Antonio Echeverría, Cujae, La Habana, Cuba

Jenny Fajardo Calderin
correo electrónico: jfajardo@ceis.cujae.edu.cu
Universidad Tecnológica de La Habana José Antonio Echeverría, Cujae, La Habana, Cuba

Danay Larrosa Uribazo
correo electrónico: dlarrosau@ceis.cujae.edu.cu
Universidad Tecnológica de La Habana José Antonio Echeverría, Cujae, La Habana, Cuba

Resumen
Diversos autores coinciden en la importancia de las pruebas como elemento de control de calidad del software y en la imposibilidad de la realización de pruebas exhaustivas. Este criterio está sustentado en que la cantidad de escenarios y valores de prueba necesarios para lograr cobertura total es grande, lo que convierte el diseño de casos de prueba y en particular la generación de sus valores en un problema combinatorio. Este trabajo presenta una propuesta para la generación automática de valores de casos de prueba funcionales, mediante el uso de algoritmos metaheurísticos, maximizando la cobertura de los escenarios. Además, se detallan los algoritmos implementados para la generación de valores iniciales y para la generación de combinaciones. Adicionalmente se describen un conjunto de buenas prácticas para utilizar el componente y la comparación de los resultados obtenidos con otras soluciones existentes.

Palabras claves: diseño de casos de pruebas, generación de valores de prueba, pruebas de software, pruebas funcionales

Recibido: 24 de marzo del 2016 Aprobado: 14 de julio del 2015

INTRODUCCIÓN

Según [1] las pruebas son muy costosas por lo que se dejan para las últimas etapas del proyecto y no se realizan con la calidad necesaria. No obstante, existen múltiples propuestas que se centran en la planificación y cálculo de los medios indispensables para realizarlas [2,3], así como a la generación automática de escenarios [4] y valores de prueba [5]. Estas propuestas persiguen el objetivo fundamental de disminuir los tiempos asociados a este proceso; simplificar su ejecución por parte de desarrolladores y probadores y alcanzar amplios grados de cobertura disminuyendo el tiempo empleado para su realización.

La generación automática de escenarios y valores de prueba es un problema combinatorio, donde intervienen un gran número de variables, por lo que dificulta su solución si se aplican técnicas tradicionales, o si la cantidad de combinaciones es tan grande e inmanejable que no se puede decidir cuáles seleccionar. En [6-8] se exploran algunas de las respuestas que brindan las temáticas de la *Ingeniería de Software Basada en Búsquedas* para dar solución a problemas combinatorios utilizando métodos de optimización [5,9]. Además, 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 [6,10-13] .

En [11,14] se describe el empleo de algoritmos de búsqueda para la generación de casos de prueba para programas orientados a objetos desarrollados en Java, estas propuestas se centran en la generación de caminos independientes, no así de los valores. En [15] se hace un recorrido por las diversas técnicas de búsqueda que se han aplicado para la generación de datos de prueba estructural [16-20]. En [21] proponen un modelo puro basado en el algoritmo *Búsqueda Tabú* para la generación automática de valores para casos de prueba. Mientras que en [22] se presenta un algoritmo memético que consiste en la fusión de una metaheurística poblacional con una lista Tabú, para gestionar el problema de la generación de caminos para casos de prueba. En [9,11,23], se propone la generación de casos de pruebas a través del empleo de heurísticas y de técnicas de ingeniería de software basadas en la búsqueda. Estas alternativas se centran en el desarrollo de valores para alcanzar un nivel de cobertura particular de los ambientes [24].

Los aportes fundamentales de las propuestas antes mencionadas están dirigidos a la utilización de algoritmos metaheurísticos y diversas modificaciones a estos algoritmos, pero no tienen en cuenta la naturaleza propia de los métodos de diseño de casos de prueba [25]. Los métodos que provienen de la disciplina de Ingeniería de Software se utilizan de forma empírica, pero no han sido incorporados a estas propuestas, lo que hace que el rango de valores que se utilizan como punto de partida para la generación de valores de prueba siga siendo grande, y por tanto el problema combinatorio continúa sin reducirse significativamente. Estos métodos de diseño tradicionales constituyen base conceptual del diseño de los casos de prueba en la Ingeniería de Software y debieran incorporarse a estas nuevas soluciones con el objetivo de reducir las combinaciones de valores a generar logrando cubrimientos similares de los escenarios de prueba.

Las propuestas existentes en generación de valores, no utilizan el hecho de que varias combinaciones de valores pueden abarcar el mismo escenario representado y por tanto este elemento podría reducir el número de combinaciones de valores, de forma tal, que se maximice la cantidad de escenarios cubiertos.

En este trabajo se presenta una propuesta para la generación de valores de pruebas funcionales, que genera valores iniciales significativos a partir de la descripción de dominio de cada variable y con ellos obtiene las combinaciones de valores que garantizan determinados niveles de cobertura de los escenarios para los casos de pruebas funcionales.

MATERIALES Y MÉTODOS

La propuesta de generador de combinaciones de valores para casos de pruebas funcionales contempla un componente de generación y combinación de valores de pruebas y su integración con otros componentes de gestión automática de casos de pruebas, conformando el CP con la información necesaria.

En la figura 1 se muestra el diagrama de actividades que describe cómo se lleva a cabo el proceso de generación y combinación de valores. El proceso comienza cuando el cliente, que puede ser un analista, programador, diseñador de pruebas u otro sistema, define el(los) dominio(s) del(los) atributo(s) que quiere generar. Una vez definidos los dominios, inicia la actividad “Generar valores iniciales para cada atributo”, de esta actividad se despliega una condicional que le permite al usuario escoger entre dos acciones. Si el usuario desea generar las combinaciones, el proceso se encamina hacia la otra actividad principal del componente llamada “Combinar valores de cada atributo”, esta actividad devuelve la lista de combinaciones de valores que el usuario almacenará posteriormente. Si el usuario no decide generar las combinaciones, el componente solo devuelve los atributos con sus valores y respectivos estados, para que el usuario pueda almacenarlo para uso posterior.

Para generar los valores iniciales o valores significativos, a partir de los cuales se generan posteriormente las combinaciones, se hace necesario que el usuario defina los dominios de cada atributo y con esta información y haciendo uso de la técnica de diseño, partición equivalente se obtienen los valores significativos para cada variable que se suministra al proceso de combinación de valores. A continuación se exponen los criterios considerados en la generación de valores significativos y los algoritmos utilizados.

Se define un algoritmo para la generación de valores iniciales de prueba para cada tipo de atributo. Este algoritmo tiene como base para la generación, la producción de valores que lleven a encontrar la mayor cantidad de no conformidades en la prueba de software.

Para asentar esta base, se hace uso de diferentes criterios en cada tipo de dato, y a partir de estos criterios se decide a qué clases de equivalencia pertenece cada valor. Se definieron seis valores significativos para el dominio númerico, hhgdfgdfhgdf

El componente *desarrollado* emplea para la obtención de los valores necesarios para un CP dos algoritmos que por su importancia en el proceso de generación serán descritos con detalle a continuación:

El **Algoritmo 1**, es el encargado de generar los valores iniciales para cada atributo, teniendo como entrada la representación de su dominio, para ello se consideran los criterios descritos. Este algoritmo discretiza los dominios, de forma tal que los valores con los que comienza el algoritmo de generación de combinaciones **(Agoritmo 2)**, reciba como entrada valores significativos para el proceso de prueba, y no todos los valores del dominio.

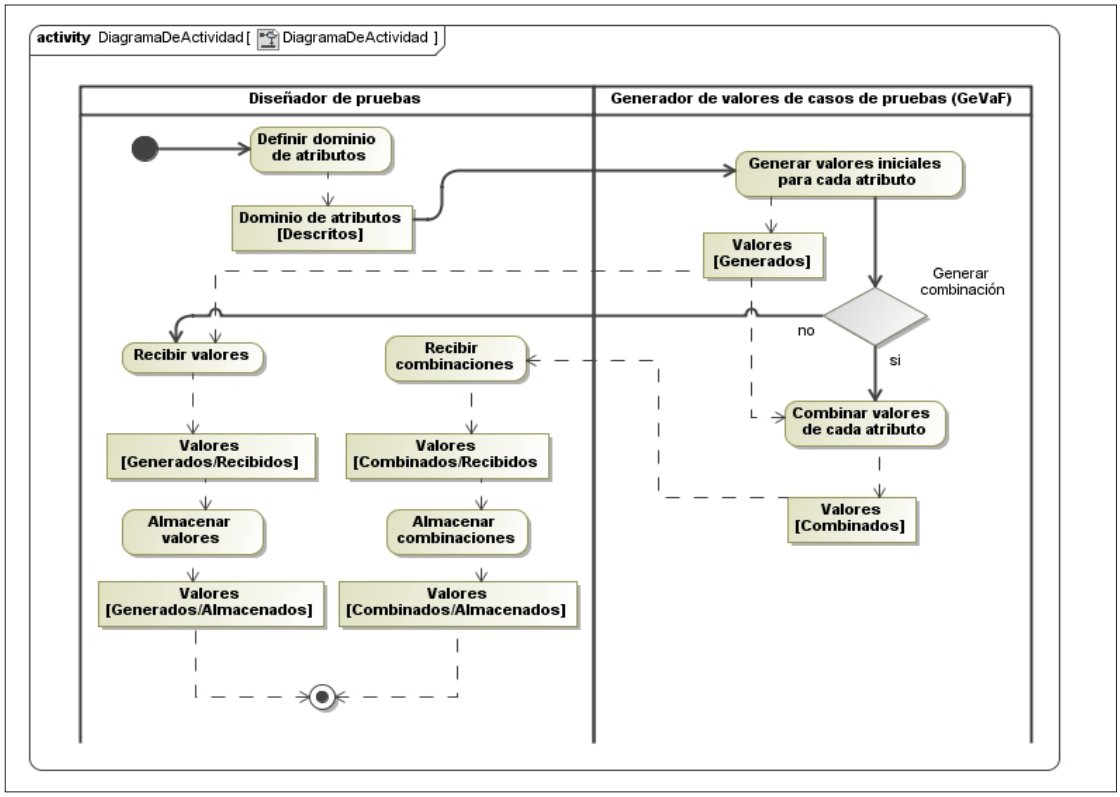

Fig. 1. Diagrama de actividades del componente de generación de valores de casos de prueba.

Algoritmo 1. Generación de valores de prueba para un atributo

Sea $\bar{\alpha} = (X_1, X, ..., X_n)$ el vector que contiene las variables o atributos de entrada a la funcionalidad que se desea probar.

Entrada: $\bar{\beta} = (y_1, y_2, ..., y_n)$ el vector que contiene la descripción del dominio de cada atributo perteneciente a $\bar{\alpha}$.

Salida: $\bar{\sigma}_k = (\lambda_1, \lambda_2, ..., \lambda_t)$ (vector que contiene los valores iniciales para la generación de combinaciones).

1. Obtener vector $\bar{\varepsilon} = (e_1, e_2, ..., e_k)$ que contiene *criterios* para instanciar valores de casos de prueba para el tipo de dato y_1.

2. $v = 1$

3. Para cada e_v, con $1 \le v \le k$

i)Obtener el valor V_i, aplicando la trasformación $T_{iv}(e_v, y_i)$ correspondiente a e_v para el atributo E_i.

ii)Obtener el valor E_i que corresponde a la clase de equivalencia de V_i

iii)Obtener el valor Z_i que corresponde a la pertenencia de V_i al dominio definido para el atributo.

iv)Construir la triada ordenada $\lambda_v = (V_i, E_i, Z_i)$

v)Adicionar λ_v a $\bar{\sigma}_i$

vi)Incrementar *v*

El **Algoritmo 2**, es el encargado de generar todas las combinaciones de valores de prueba para probar una funcionalidad, teniendo en cuenta los valores de cobertura especificados.

Algoritmo 2. Generación de combinaciones de valores para un requisito funcional

Entrada: $\bar{\alpha} = (X_1, X, ..., X_n)$ (vector que contiene los atributos de los cuales se quieren combinar sus valores). $\phi = (\bar{\sigma}_1, \bar{\sigma}_2, ..., \bar{\sigma}_n)$ (vector que contiene n vectores con los valores iniciales de cada atributo $\bar{\sigma}_i$ de la funcionalidad que se quiere probar, donde el vector $\bar{\sigma}_i$ tiene la forma $\sigma_i = (\lambda_1, \lambda_2, ..., \lambda_m)$ donde la longitud m del vector $\bar{\sigma}_i$ puede ser diferente para cada $\bar{\phi}$ del vector) $\bar{\phi}$

Salida: M_{lxn} (matriz que contiene las combinaciones de valores generados).

1) Para cada atributo $_{\bar{\alpha}=(X_1,X,...,X_n)}$ de $\bar{\alpha} = (X_1, X, ..., X_n)$:

i)Si no está generado.

a)Generar el vector $\bar{\sigma}_i = (\lambda_1, \lambda_2, ..., \lambda_t)$ con los valores iniciales del atributo utilizando el Algoritmo 1.

b) Adicionar $\bar{\sigma}_i = (\lambda_1, \lambda_2, ..., \lambda_t)$ al vector $\bar{\phi} = (\bar{\sigma}_1, \bar{\sigma}_2, ..., \bar{\sigma}_n)$ en la posición l.

ii) incrementar *i*.

2.Calcular cantidad total de iteraciones l a partir de la cobertura C.

3.Calcular cantidad máxima de combinaciones de clases de equivalencia $(\prod_{i=1}^{n} longitud(\bar{\varepsilon}_i))$ según se indica en el modelo de optimización.

4. *j*=1

5. Para cada iteración *j* < *l*

i)Generar una combinación de valores $\gamma_i = (\varphi_1, \varphi_2, ..., \varphi_n)$ en la que cada elemento φ_i es un par ordenado $\varphi_i = (V_i, E_i)$ donde V_i es el valor generado para el atributo E_i y E_i es el valor de la clase de equivalencia correspondiente.

ii)Validar la combinación $\gamma_j = (\varphi_1, \varphi_2, ..., \varphi_n)$, aplicando los mecanismos de penalizacia según el modelo propuesto.

iii)Si la combinación γ_j es válida

a)Calcular heurística $fh(\bar{\gamma}_j)$ según la ecuación presentada.

b)Si $fh(\bar{\gamma}_j) \geq umbral(\bar{\gamma}_j)$ entonces adicionar a M_{lxn}.

c)Si $j \geq \left(\prod_{i=1}^{n} longitud(\bar{\varepsilon}_i)\right)$ recalcular el umbral con la expresión presentada.

iv)Incrementar j

El componente GeVaF, que implementa esta propuesta ofrece una solución que resuelve la situación problemática de tres formas distintas y de fácil empleo para el usuario que la utilice.

La primera variante es utilizando este componente en otra aplicación que capture los dominios de cada atributo a través de una interfaz de usuario con el objetivo de evitarles a los futuros interesados la obligación de consumir un servicio. De esta forma se podrá obtener las combinaciones de valores para casos de pruebas funcionales ocultando tras la interfaz todo el procesamiento de ficheros y variables correspondientes.

La segunda variante es utilizar el componente haciendo uso de su capa de servicios para el consumo de los usuarios con el principal objetivo de facilitar la integración con cualquier tipo de aplicación que esta ofrece, ya sea *desktop* o *web*.

Se decidió además darle solución a este problema usando una variante de componente que permita tratarlo como una biblioteca, para darle al usuario una forma de solución arquitectónica distinta y poder ajustar la solución a cualquier necesidad.

Los resultados de las combinaciones generadas se almacenan en un fichero XML. En la figura 2 se muestra un ejemplo de una combinación de los atributos “Usuario” y “Contraseña”:

RESULTADOS Y DSICUSIÓN

Para validar el componente de generación de valores de prueba GeVaF se realizaron varias ejecuciones con el objetivo de comparar los resultados con otras propuestas nacionales.

Se diseñó un experimento en el que se fija el número de iteraciones (I) a partir del 100 % de cobertura (C) de los valores iniciales X_i. Para ejecutar el experimento se utilizó como caso de estudio una aplicación real, con el objetivo de verificar si se reduce la cantidad de combinaciones de valores para casos de pruebas funcionales generadas por GeVaF evidenciando que se cubra el 100 % de los escenarios, con respecto a las combinaciones generadas por el componente que solo utiliza metaheurísticas. En este caso de estudio se diseñaron las pruebas para seis de sus funcionalidades que, como promedio tienen tres atributos por cada una de ellas.

```
        ="xs:string">@</Valor>
        <estado>true</estado>
        <ClaseDeEquivalencia xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="
        http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">13</ClaseDeEquivalencia>
    </item>
    <item>
        <Nombre>Contraseña</Nombre>
        <Valor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type
        ="xs:string">23 </Valor>
        <estado>true</estado>
        <ClaseDeEquivalencia xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="
        http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">23</ClaseDeEquivalencia>
    </item>
</listaCombinaciones>
<listaCombinaciones>
    <item>
        <Nombre>Usuario</Nombre>
        <Valor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type
        ="xs:string"></Valor>
        <estado>false</estado>
        <ClaseDeEquivalencia xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="
        http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">12</ClaseDeEquivalencia>
    </item>
    <item>
        <Nombre>Contraseña</Nombre>
        <Valor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type
        ="xs:string">@</Valor>
        <estado>true</estado>
```

Fig. 2. Resultado de una combinación de valores generada con GeVaF.

En la tabla 1 se muestra la información referente a los casos de estudio, señalando para cada uno la cantidad de variables, la cantidad de escenarios y la cantidad máxima de combinaciones necesarias para cubrir los escenarios correspondientes a cada funcionalidad a probar, así como los resultados obtenidos con la ejecución de la herramienta GeVaF, que utiliza metaheurísticas y la técnica de partición equivalente de la disciplina Ingeniería de Software y su comparación con un componente que solo utiliza metaheurísticas. Para ejecutar este caso de estudio, ambas propuestas se realizaron ejecutando el algoritmo de "Búsqueda Aleatoria".

Tabla 1
Resultados de ejecuciones

Cant. variab.	Escenarios	Combinaciones máximas	Metaheurísticas y técnicas de Ing. de Software	Metaheurísticas
2	9	81	9	5 (55 %)
3	27	27	27	7 (25,9 %)
2	12	27	12	6 (50 %)
4	108	504	108	16 (14,8 %)
2	9	28	9	5 (55 %)
3	27	112	27	7 (25,9 %)

Como se puede observar al utilizar metaheurísticas y la técnica de Ingeniería de Software se genera al menos una combinación de valores para probar cada uno de los escenarios, por tanto se cubre el 100 % de los escenarios con una cantidad de combinaciones similar a las obtenidas en el otro algoritmo. En contraste las propuestas de optimización que solo utilizan metaheurísticas generan como promedio el 37,7 % de los escenarios. Es necesario aclarar que en el experimento se emplearon combinaciones de variables que incluyen datos del tipo fecha, numérico, cadena, lógico y enumerado.

CONCLUSIONES

En este trabajo se definieron las clases de equivalencia que permiten cubrir las pruebas para diferentes tipos de atributos, y las técnicas de diseño de pruebas funcionales que permiten lograr una mayor detección de errores según la bibliografía consultada.

Se diseñaron casos de estudio para mostrar los diferentes usos del componente, lo que facilita su aplicación a diferentes contextos de diseño de los casos de pruebas.

Se presentan los resultados de un experimento que demuestra las ventajas que puede ofrecer el componente desarrollado en cuanto a la maximización de escenarios cubiertos con las combinaciones de valores de prueba generados.

REFERENCIAS

1. **MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom.** *The art of software testing*, John Wiley & Sons, 2011.
2. **LAMANCHA PÉREZ, Beatriz; POLO, Macario,** "Generación automática de casos de prueba para líneas de producto de software". *Innovación*, *Calidad e Ingeniería del Software*. 2009, núm. 5, pp. 17.
3. **MEMON, Atif M.; POLLACK, Martha E.; SOFFA, Mary Lou,** "Hierarchical GUI test case generation using automated planning". *Software Engineering*, IEEE Transactions on. 2001, núm. 27, pp. 144-155.
4. **BOUQUET, Fabrice; GRANDPIERRE, Christophe** *et. al.* "A test generation solution to automate software testing", *Proceedings of the 3rd international workshop on Automation of software test*, ACM, 2008, pp. 45-48.
5. **ANAND, Saswat; BURKE, Edmund K.** *et. al*. "An orchestrated survey of methodologies for automated software test case generation", *Journal of Systems and Software*. 2013, núm. 86, pp. 1978-2001.
6. **HARMAN, Mark; MANSOURI, S. Afshin; ZHANG, Yuanyuan.** "Search-based software engineering: Trends, techniques and applications", ACM Computing Surveys (CSUR). 2012, núm. 45, pp. 11.
7. **SCHWARZL, Christian; PEISCHL, Bernhard.** "Generation of executable test cases based on behavioral UML system models", *Proceedings of the 5th Workshop on Automation of Software Test*, ACM.2010, pp. 31-34.
8. **POLO USAOLA, Macario; REALES MATEO, Pedro.** "Enseñanza de la mutación en pruebas de software", Jornadas de Enseñanza de la Informática (18es: 2012: Ciudad Real), 2012.
9. **HARMAN, Mark.** "Automated test data generation using search based software engineering". *Automation of Software Test*, AST'07. Second International Workshop on, IEEE, 2007, p. 2.
10. **IQBAL, Muhammad Zohaib; ARCURI, Andrea; BRIAND, Lionel.** "Empirical investigation of search algorithms for environment model-based testing of real-time embedded software". *Proceedings of the 2012*

International Symposium on Software Testing and Analysis. ACM. 2012, pp. 199-209.

11. **RIBEIRO BREGIEIRO, José Carlos.** "Search-based test case generation for object-oriented java software using strongly-typed genetic programming", *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*. ACM. 2008, pp. 1819-1822.
12. **ZHIQIANG ZHANG, Jun Yan; Yong Zhao, Jian Zhang.** "Generating combinatorial test suite using combinatorial optimization", *The Journal of Systems and Software,* 2014.
13. **TING CHEN, Xiao-song Zhang; Shi-ze Guo, Hong-yuan Li; Yue Wu.** *State of the art: Dynamic symbolic execution for automated test generation*: Future Generation Computer Systems. 2013.
14. **AHMED, Bestoun S.; ZAMLI, Kamal Z.** "Comparison of metahuristic test generation strategies based on interaction elements coverage criterion", *Industrial Electronics and Applications* (ISIEA). IEEE Symposium on. IEEE, 2011, pp. 550-554.
15. **VARSHNEY, Sapna; MEHROTRA, Monica**. "Search based software test data generation for structural testing: a perspective", *ACM SIGSOFT Software Engineering Notes*. 2013, núm. 38, pp. 1-6.
16. **FERGUSON, Roger; KOREL, Bogdan**. *The chaining approach for software test data generation*, ACM Transactions on Software Engineering and Methodology (TOSEM). 1996, núm. 5, pp. 63-86.
17. **MICHAEL, Christoph C.; MCGRAW, Gary; SCHATZ, Michael A.** "Generating software test data by evolution". *Software Engineering*, IEEE Transactions on. 2001, núm. 27, pp. 1085-1110.
18. **PARGAS, Roy P.; HARROLD, Mary Jean; PECK, Robert R.** "Test-data generation using genetic algorithms", *Software Testing Verification and Reliability*. 1999, núm. 9, pp. 263-282.
19. **WEGENER, Joachim; BARESEL, André; STHAMER, Harmen.** "Evolutionary test environment for automatic structural testing", *Information and Software Technology*. 2001, núm. 43, pp. 841-854.
20. **PACHAURI, Ankur; SRIVASTAVA, Gursaran.** "Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism". *Journal of Systems and Software*. 2013, núm. 86, pp. 1191-1208.
21. **DÍAZ, Eugenia; TUYA, Javier** *et. al***.** "A tabu search algorithm for structural software testing", *Computers & Operations Research*. 2008, núm. 35, pp. 3052-3072.
22. **LANZARINI, Laura Cristina; BATTAIOTTO, Pedro Eduardo.** "Dynamic generation of test cases with metaheuristics", *Journal of Computer Science & Technology*. 2010, núm. 10.
23. **SAGARNA, Ramón; MENDIBURU, Alexander** *et. al***.** "Assisting in search heuristics selection through multidimensional supervised classification: A case study on software testing". *Information Sciences.* 2014.
24. **SAKTI, Abdelilah; GUÉHÉNEUC, Yann-Gaël; PESANT, Gilles.** "Boosting search based testing by using constraint based testing", *Search Based Software Engineering*, Springer. 2012, pp. 213-227.
25. **PRESSMAN, R.** *Ingeniería del software. Un enfoque práctico*. Sexta edición. McGraw-Hill, Interamericana Editores, SA de CV México. 2005.

AUTORES

Arloys Macías Rojas
Ingeniero Informático, Departamento de Informatización, Facultad de Ingeniería Informática, Universidad Tecnológica de La Habana José Antonio Echeverría, Cujae, La Habana Cuba

Martha Dunia Delgado Dapena
Ingeniera Informatica, Doctora en Ciencias Técnicas, Profesora Titular, Facultad de Ingeniería Informática, Universidad Tecnológica de La Habana, José Antonio Echeverría, Cujae, La Habana, Cuba

Jenny Fajardo Calderín
Ingeniera Informática, Doctora en Ciencias Técnicas, Profesora Asistente, Departamento de Inteligencia Artificial e Infraestructura de Sistemas Informáticos, Facultad de Ingeniería Informática, Universidad Tecnológica de La Habana, José Antonio Echeverría, Cujae, La Habana, Cuba

Danay Larrosa Uribazo
Ingeniera Informática, Instructora, Departamento de Ingeniería de Software, Facultad de Ingeniería Informática, Universidad Tecnológica de La Habana josé Antonio Echeverría, Cujae, La Habana, Cuba

Automatic Generation of combination of Values for Functional Testing Using Metaheuristics

Abstract

Several authors agree with the importance of the tests like element of quality control of the software and in the impossibility of their realization of exhaustive way. This opinion defends that, the necessary quantity of stages and test values to achieve the maximum coverage is too big, what converts the test-case design, and in particular the generation of its values, in a combinatorial problem. That´s why, in many instances, in front of the impossibility of covering all the stages, testers leave out of the design some interesting values, which can discover inconsistencies with the specified requirements.This work presents a proposal for the automatic generation of values of functional test cases, by means of the use of meta-heuristic algorithms and maximizing the coverage of the stages. Furthermore, the algorithms implemented for the generation of initial values and for the generation of combinations are detailed. Additionally a set of good practices to use the component and the comparison of the obtained results with other existing solutions are described.

Key words: test cases design, test values generation, software test, functional test