Lámpsakos | N°.15 | pp. 51-58 | enero-junio | 2016 | ISSN: 2145-4086 | Medellín–Colombia

Generador de Valores de Casos de Prueba Funcionales

Generator of Values for Functional Test Cases

Arloys Macías-Rojas, Ing.
Centro Universitario José Antonio Echeverría
La Habana, Cuba
amacias@ceis.cujae.edu.cu

Martha Dunia Delgado-Dapena, PhD.
Centro Universitario José Antonio Echeverría
La Habana, Cuba
marta@ceis.cujae.edu.cu

Jenny Fajardo-Calderín, MSc.
Centro Universitario José Antonio Echeverría
La Habana, Cuba
jfajardo@ceis.cujae.edu.cu

Danay Larrosa-Uribazo, Est.
Centro Universitario José Antonio Echeverría
La Habana, Cuba
dlarrosau@ceis.cujae.edu.cu

(Recibido el 20-10-2015. Aprobado el 16-12-2015)

Estilo de Citación de Artículo:
A. Macías-Rojas, M.D Delgado-Dapena, J. Fajardo-Calderín, D. Larrosa-Uribazo "Generador de Valores de Casos de Prueba Funcionales", *Lámpsakos*, no. 15, pp 51 - 58, 2016.

Resumen. Diversos autores coinciden en la importancia de las pruebas como elemento de control de calidad del software y en la imposibilidad de 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 clave: Diseño de casos de pruebas; Generación de valores de prueba; Pruebas de software; pruebas funcionales.

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.

Keywords: Test Cases Design; Test Values Generation; Software Test; Functional Test.

1. INTRODUCCIÓN

Según [13] 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 [11], [15], así como en la generación automática de escenarios [3] y valores de prueba [2]. 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. No obstante, el número de combinaciones continúa sin reducirse significativamente, debido a que no se incorporan técnicas que permitan reducir el espacio de búsqueda, o criterios provenientes de la disciplina de Ingeniería de Software que reduzcan el espacio de combinaciones resultantes.

Para resolver la problemática planteada los autores del presente trabajo llevan a cabo el proyecto Testeo Temprano, en el que se intenta maximizar cobertura de escenarios con el menor número de combinaciones de valores posibles. Para ello, se generan combinaciones de valores que cubren las combinaciones de particiones equivalentes asociadas con cada escenario diferente de la funcionalidad que se intenta probar.

La hipótesis de este trabajo se basa en que la implementación de mecanismos automáticos que combinen el uso de algoritmos heurísticos con técnicas tradicionales de diseño de casos de prueba, permite obtener al menos una combinación de valores para cada escenario de prueba, según el nivel de cobertura de escenarios especificado.

En este artículo se presenta el componente de software GeVaF, el cual automatiza una propuesta que combina algoritmos heurísticos con la técnica de diseño de casos de pruebas de particiones equivalentes. Se describen las características generales de GeVaF y una comparación con el uso de métodos heurísticos sin las técnicas de diseño provenientes de la Ingeniería de Software.

2. MARCO TEÓRICO

La generación automática de escenarios y valores de prueba es un problema combinatorio, donde intervienen un gran número de variables, por lo que se dificulta su solución cuando se aplican técnicas tradicionales, o si la cantidad de combinaciones es tan grande e inmanejable que no se puede decidir cuáles seleccionar. En [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 [2], [7]. 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 [4], [8], [9].

En [4] y [1] 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 [18] se hace un recorrido por las diversas técnicas de búsqueda que se han aplicado para la generación de datos de prueba estructural [6], [12], [14], [19]. En [5] proponen un modelo puro basado en el algoritmo “Búsqueda Tabú” para la generación automática de valores para casos de prueba. Mientras que en [10] se presenta una solución basada en, la fusión de una metaheurística poblacional con una lista Tabú, lo que se conoce como un algoritmo memético, para gestionar el problema de la generación de caminos para casos de prueba. En [4], [7] 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 [17].

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 [16]. Estos 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 utiliza como punto de partida para la generación de valores de prueba siga siendo grande, y por tanto el problema combinatorio continúe sin reducirse

significativamente. Estos métodos de diseño tradicionales constituyen la 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 tampoco 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.

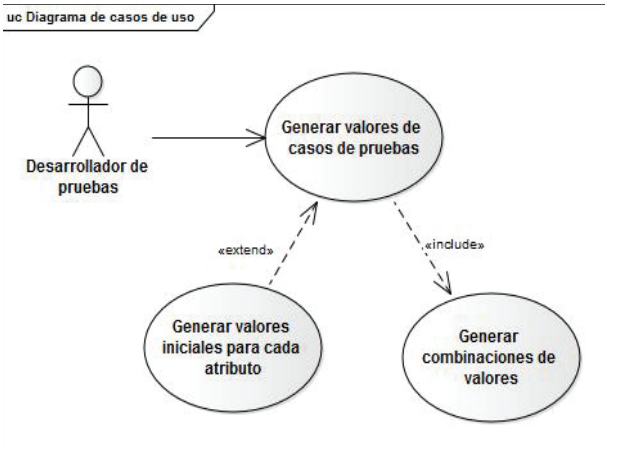

Fig. 1 Diagrama UML con los casos de uso de GeVaF.

3. METODOLOGÍA DE CONSTRUCCIÓN DEL GENERADOR DE VALORES DE PRUEBAS

La propuesta del artículo automatiza un modelo de optimización que combina el uso de algoritmos heurísticos con la técnica de diseño de casos de prueba de particiones equivalentes. Para ello se diseña e implementa el componente GeVaF y posteriormente, se comparan los resultados obtenidos contra los casos de prueba sin las técnicas de Ingeniería de Software.

El objetivo fundamental de GeVaF es brindar las combinaciones posibles de valores de prueba de manera reducida utilizando metaheurísticas, a partir de las características de los atributos.

3.1. Generador de valores de pruebas

El Generador de Valores de Pruebas (GeVaF), genera los valores a partir de las características de los atributos, dadas por el usuario, miembro del equipo de proyecto, en el momento de realizar la descripción detallada correspondiente a un requisito funcional que deriva en un caso de uso del sistema. En la fig. 1 se muestra el diagrama de casos de uso de GeVaF.

El enfoque utilizado para la estructuración en capas del componente desarrollado está basado en reutilización, y se ilustra en la fig. 2. La vista de la arquitectura se encuentra proyectada en dos capas: capa específica y capa general.

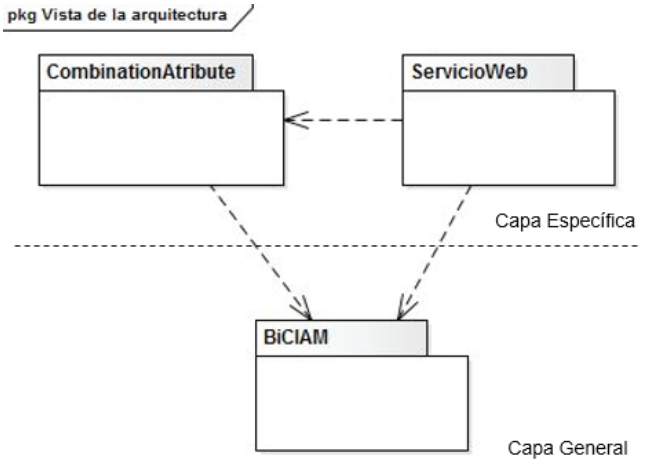

Fig. 2 Diagrama UML con la Vista de Arquitectura de la solución

3.2. Elementos del modelo

Capa Específica: se muestran los elementos (paquetes) del sistema actual, que no son reutilizados. A continuación se muestra una breve explicación de su contenido:

CombinationAtribute. Contiene las diferentes clases que logran la definición del problema, como “CombinacionesMutationOperator” que contiene los operadores específicos del problema; “Atributo” que contiene el nombre, valor y estado de un atributo; “AtributoValores” que se encarga de las funcionalidades referentes a los atributos y sus valores posibles; “Combinaciones” que define la cantidad de variables

y combinaciones a crear, y “ProblemaDatos” que controla las operaciones referentes a la lista de “AtributoValores” que posee.

ServicioWeb. Se encarga de la codificación y configuración de la capa de servicio del componente y entre sus principales clases se encuentra, “Service” que maneja la lista de combinaciones y todos los métodos del servicio web necesarios para devolver lo que se quiere mostrar cuando se consuma el servicio; “ServiceExponer” que es donde están los métodos que se van a exponer en dicho servicio a la hora de consumirlo, y la clase “Tester” que se encarga de la configuración del problema, y de los algoritmos que se vayan a ejecutar desde BiCIAM.

Adicionalmente se implementó una aplicación cliente que, aunque no forma parte de la solución propuesta, consume los servicios definidos en el paquete ServicioWeb. Esta aplicación se creó con el objetivo de simular el comportamiento de un cliente que pretende consumir el ServicioWeb mencionado con anterioridad, y así comprobar y validar los resultados obtenidos en dicho servicio.

Dentro de estos paquetes se encuentran las clases que abstraen el problema del algoritmo, permitiendo que estas sean empleadas para su definición. Todos los elementos que se encuentran en esta capa cumplen con la cualidad de ser específicos al problema de generar las combinaciones y de no ser reutilizables para solucionar otros problemas.

Capa General: se muestran los elementos de cualquier nivel de responsabilidad (al igual que en la capa anterior) pero que son reutilizados. Entre los elementos se encuentra BiCIAM que es la biblioteca de clases que implementa algoritmos metaheurísticos basada en un modelo unificado de estos algoritmos.

Para una mayor comprensión del funcionamiento del componente GeVaF, en la fig. 3 se muestra un diagrama de flujo con las principales actividades que se realizan y los objetos necesarios durante su ejecución.

Con la culminación de este componente se puede contar con una solución que resuelve la situación problemática de tres formas distintas y de fácil uso 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

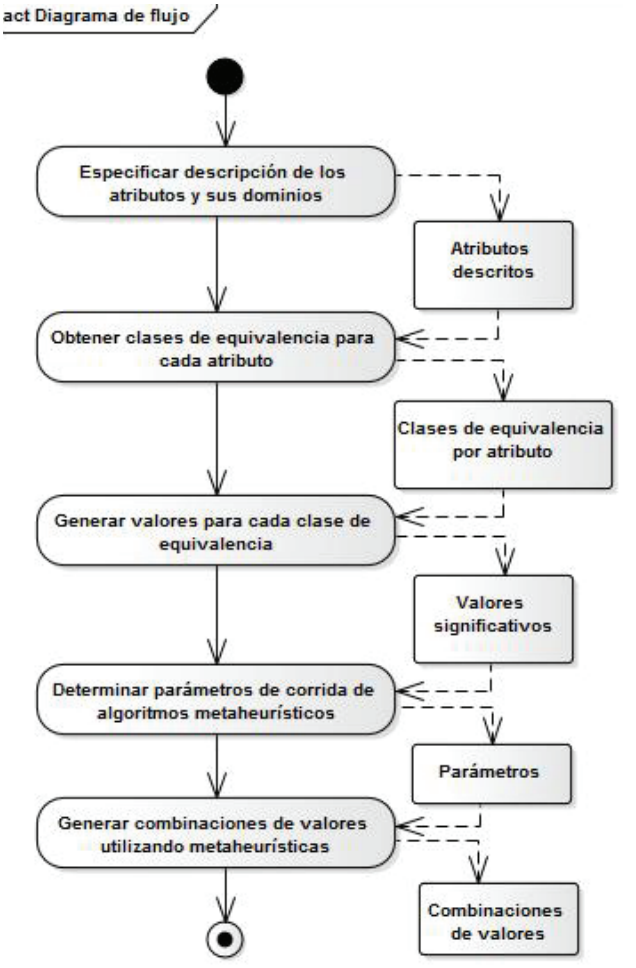

Fig. 3 Diagrama de flujo del componente GeVaF

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.

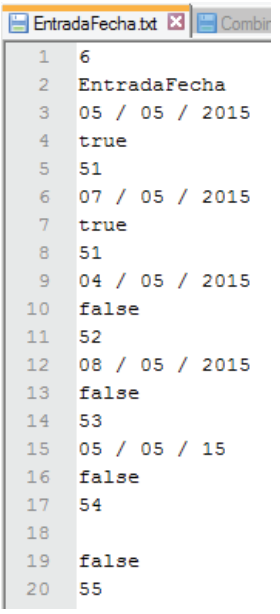

Fig. 4 Fichero generado por GeVaF

3.3. Caracterización del diseño

El diseño es intuitivo, de forma que permite caracterizar a los atributos con la mayor facilidad posible; el producto puede integrarse con cualquier componente de gestión de atributos desarrollado en Java y las descripciones de las variables y sus dominios, suministradas por el diseñador de casos de pruebas, se brindarán a través de ficheros textos con la siguiente estructura: “Cantidad de valores generados para ese atributo”; “Nombre del fichero o atributo”; “Nombre del primer valor generado”; “Estado del valor generado” (valor de verdad en correspondencia con la descripción que detalló el usuario); “Número que representa la clase de equivalencia correspondiente a dicho valor con respecto a la descripción del atributo detallada por el usuario que lo creó”. En la fig. 4 se refleja un ejemplo de un fichero generado por GeVaF:

Los resultados de las combinaciones generadas se almacenan en un fichero XML. En la fig. 5 se muestra un ejemplo de una combinación de los atributos “Usuario” y “Contraseña”:

Fig. 5 Resultado de una combinación de valores generada con GeVaF

3.4. Resultados y discusión

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

Se diseñó un experimento en el que se fija el número de iteraciones a partir del 100% de cobertura de los valores iniciales. Para ejecutar el experimento se utilizó como caso de estudio una aplicación real, en la que se diseñaron las pruebas para seis de sus funcionalidades que, como promedio tiene tres atributos por cada una de ellas. Las variables incluidas cubren los tipos de datos: cadena, numérico, enumerado, lógico y fecha.

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 necesaria 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 particiones equivalentes de la disciplina Ingeniería de Software y su comparación con un componente que solo utiliza metaheurísticas. Las ejecuciones para este caso de estudio, ambas propuestas, se realizaron ejecutando el algoritmo de “Búsqueda Aleatoria”.

Como se puede observar en la Tabla 1, al utilizar metaheurísticas y técnicas 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.

La culminación de este trabajo deja un campo futuro para continuar su desarrollo, por tanto se recomienda para dichos trabajos la consideración de: otros tipos de datos, como pueden ser los conjuntos; tener en cuenta la referencia cruzada entre los atributos o variables de una misma funcionalidad del proyecto; y además contar con otros algoritmos heurísticos como los poblacionales.

4. CONCLUSIONES

En este trabajo se definieron las clases de equivalencia que permiten cubrir las pruebas para diferentes tipos de atributos, a partir de la identificación de los tipos de atributos más empleados en los entornos de desarrollo, y las técnicas de diseño de pruebas funcionales que permiten lograr una mayor detección de errores según la bibliografía consultada.

Se construyó un componente orientado a servicios que integra algoritmos de generación de valores iniciales y utiliza metaheurísticas para generar las combinaciones de valores.

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 muestran 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.

Tabla 1.
Resultados de Ejecuciones

Cantidad de variables	Escenarios cubiertos	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%)

REFERENCIAS

[1] B. S. Ahmed, and K. Z. Zamli, “Comparison of metahuristic test generation strategies based on interaction elements coverage criterion,” en 2011 *IEEE Symposium on Industrial Electronics and Applications (ISIEA), Langkawi*, pp. 550-554, 2011. Available: DOI: 10.1109/ISIEA.2011.6108773.

[2] S. Anand, E. K. Burke, J. C. Tsong Yueh Chenc, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An orchestrated survey of methodologies for automated software test case generation,” *The Journal of Systems and Software,* vol. 86, no. 8, pp. 1978-2001, 2013. ISSN: 0164-1212. Available: doi: 10.1016/j.jss.2013.02.061.

[3] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux, “A test generation solution to automate software testing,” en *Proceedings of the 3rd international workshop on Automation of software test Leipzig, Germany, ACM,* pp. 45-48, 2008. Available: doi: 10.1145/1370042.1370052.

[4] J. C. Bregieiro, “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 Atlanta, Georgia, ACM,* pp. 1819-1822, 2008. Available: doi: 10.1145/1388969.1388979.

[5] E. Díaz, J. Tuya, R. Blanco, and J. J. Dolado, "A tabu search algorithm for structural software testing," *Computers & Operations Research,* vol. 35, no. 10, pp. 3052-3072, 2008. ISSN: 0305-0548. Available: doi: 10.1016/j.cor.2007.01.009.

[6] R. Ferguson, and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 5, no. 1, pp. 63-86, 1996. ISSN: 1049-331X. Available: doi: 10.1145/226155.226158.

[7] M. Harman, "Automated test data generation using search based software engineering," en *Second International Workshop on Automation of Software Test, Minneapolis, Minnesota,* p. 2, 2007. Available: doi: 10.1109/AST.2007.4.

[8] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR),* vol. 45, no. 1, p. 11, 2012. ISSN: 0360-0300. Available: doi: 10.1145/2379776.2379787.

[9] M. Z. Iqbal, A. Arcuri, and L. Briand, "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 Minneapolis, Minnesota, ACM,* pp. 199-209, 2012. Available: doi: 10.1145/2338965.2336777.

[10] L. C. Lanzarini, and P. E. Battaiotto, "Dynamic generation of test cases with metaheuristics," *Journal of Computer Science & Technology, v*ol. 10, no. 2, p. 91, 2010. ISSN: 1860-4749. [Online] Disponible: http://sedici.unlp.edu.ar/bitstream/handle/10915/21338/Documento_completo.pdf?sequence=1. Consultado: 02-10-2015.

[11] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Transactions on Software Engineering,* vol. 27, no. 2, pp. 144-155, 2001. ISSN: 0098-5589. Disponible: doi: 10.1109/32.908959.

[12] C. C. Michael, and G. McGraw, "Generating software test data by evolution," *IEEE Transactions on Software Engineering,* vol. 27, no. 12, pp. 1085-1110, 2001. ISSN: 0098-5589. Available: doi: 10.1109/32.988709.

[13] G. J. Myers, T. Badgett, and C. Sandler, *The art of software testing*, 3a. ed., New Jersey, USA: JohnWiley & Sons, 2011. ISBN: 978-1-118-03196-4. [Online] Disponible: http://www.computing.dcu.ie/~ray/teaching/CA358/TheArtofSoftwareTesting.pdf. Consultado: 07-10-2015.

[14] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability,* vol. 9, no. 4, pp. 261-313, 1999. ISSN: 1099-1689. [Online] Disponible: http://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Readings/pga.pdf.

[15] B. Pérez, and M. Polo, "Generación automática de casos de prueba para Líneas de Producto de Software," *REICIS. Revista Española de Innovación, Calidad e Ingeniería del Software,* vol. 5, no. 2, pp. 17-27, 2009. ISSN: 1885-448. [Online] Disponible en: http://www.redalyc.org/comocitar.oa?id=92217153004.

[16] R. S. Pressman, *Ingeniería del Software: Un enfoque práctico*, 7a. ed., México D.F.: McGraw-Hill, 2010. ISBN: 978-607-15-0314-5. [Online] Disponible en: http://es.slideshare.net/jes4791/ingenieria-del-software-un-enfoque-practico. Consultado: 09-10-2015.

[17] A. Sakti, Y.-G. Guéhéneuc, and G. Pesant, "Boosting search based testing by using constraint based testing," *Search Based Software Engineering*, Berlin, Germany: Springer-Verlag Berlin Heidelberg, pp. 213-227, 2012. ISBN: 978-3-642-33118-3. Available: doi: 10.1007/978-3-642-33119-0_16.

[18] S. Varshney, and M. Mehrotra, "Search based software test data generation for structural testing: a perspective," *ACM SIGSOFT Software Engineering Notes,* vol. 38, no. 4, pp. 1-6, 2013. ISSN: 0163-5948. Disponible: doi: 10.1145/2492248.2492277.

[19] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology,* vol. 43, no. 14, pp. 841-854, 2001. ISSN: 0950-5849. Available: doi: 10.1016/S0950-5849(01)00190-2.