Modelo para la generación automática de pruebas tempranas basadas en búsquedas

Martha Dunia Delgado Dapena, Arloys Macías Rojas, Danay Larrosa Uribazo,
Sandra Verona Marcos, Perla Beatriz Fernández Oliva

Universidad Tecnológica de La Habana "José Antonio Echeverría",
Facultad de Ingeniería Informática, CUJAE, La Habana,
Cuba

{marta, amacias, dlarrosau, sverona, perla}@ceis.cujae.edu.cu

Resumen. En el presente trabajo se presenta una propuesta para la generación de pruebas tempranas de *software*. La propuesta incluye las actividades para la ejecución de pruebas en un entorno productivo, así como el modelo de optimización que reduce la cantidad de casos de prueba funcionales considerando criterios de cobertura de escenario y utilizando algoritmos de búsqueda heurística. Adicionalmente el modelo contiene un conjunto de herramientas basadas en búsqueda que dan soporte a la generación de casos de pruebas funcionales.

Palabras clave. Pruebas funcionales, generación automática de casos de prueba, algoritmos metaheurísticos.

Model for Automatic Generation of Search-based Early Tests

Abstract. This article presents a proposal for the generation of software early tests. It includes: activities for test execution in a production environment, as well as the optimization model that reduces the number of functional test cases considering scenario covering criteria and using heuristic search algorithms. In addition, the model contains a collection of search-based tools that support the generation of functional test cases.

Keywords. Functional tests, automatic generation of test cases, metaheuristics algorithms.

1. Introducción

Según [20] las pruebas de *software* son muy costosas, por lo que se dejan para las últimas etapas del proyecto y, por ello, típicamente no se realizan con la calidad necesaria. No obstante, en el estado del arte existen múltiples propuestas que se centran en la planificación y cálculo de los medios indispensables para realizarlas [18,15], así como a la generación automática de escenarios [3] y valores de prueba [4]. 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, a la vez, el tiempo empleado para su realización.

Las pruebas de software continúan ocupando espacios importantes en los trabajos científicos de múltiples investigadores; en particular, se mantienen como problemas abiertos la generación de caminos y valores de pruebas para apoyar el diseño de los casos de prueba [5,13, 21, 24,27], así como los procesos vinculados con las pruebas de software [27, 6, 8, 25, 30, 31].

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 [12] 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 [4, 11]. Además, existen trabajos recientes que automatizan la realización de las pruebas de software con respecto a la generación

ISSN 1405-5546
doi: 10.13053/CyS-21-3-2716

de escenarios y valores de prueba, utilizando técnicas para evadir la explosión combinatoria [12, 4,14].

En [4, 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 [28] se hace un recorrido por las diversas técnicas de búsqueda que se han aplicado para la generación de datos de prueba estructural [9,19, 22, 29]. En [7] los autore4s proponen un modelo puro basado en el algoritmo "Búsqueda Tabú" para la generación automática de valores para casos de prueba. Mientras que en [16] 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 [11, 4] 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 [26].

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 [23]. 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 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 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 mencionadas anteriormente van desde la utilización de algoritmos de optimización e inteligencia artificial para resolver el problema de la explosión combinatoria de los caminos y valores de pruebas, hasta propuestas de *frameworks* para la automatización de algunos elementos del proceso. En este último caso las propuestas son prototipos para validar la solución teórica, pero no se han incorporado a las soluciones comerciales los elementos de generación de los casos de prueba, de forma tal que puedan ser utilizados por desarrolladores y equipos de probadores, reduciendo así el esfuerzo vinculado con esta actividad de diseño que es altamente costosa.

Este artículo presenta una propuesta para ejecutar las pruebas de *software* en diferentes momentos del desarrollo del producto de *software*, que es independiente del enfoque o la metodología de desarrollo que se utilice en el proceso productivo: El Modelo de Ejecución de Pruebas Tempranas Basada en Búsquedas (MTest.search). En el trabajo se hace énfasis en la generación automática de las pruebas funcionales, por lo que se presenta un conjunto de herramientas automatizadas que permiten soportar el modelo en el entorno productivo.

2. Materiales y Métodos

El modelo, MTest.search, consta de Flujos de Trabajo para la ejecución de pruebas tempranas en el entorno de producción, Modelos de Optimización para reducción de casos de pruebas funcionales y unitarias, y Herramientas Automatizadas Integradas que dan soporte a la ejecución de los flujos de trabajo.

Para la ejecución de la propuesta se recomienda tener una estructura en el proyecto con al menos dos personas que desempeñen las funciones del ingeniero de prueba, que de conjunto con clientes y/o desarrolladores realicen el diseño de las pruebas y definan cuáles son los perfiles de prueba para cada proyecto. Además, quienes desempeñen este rol deben encargarse de mantener la implementación de las pruebas automáticas si fuera necesario.

El modelo contempla un grupo de flujos de trabajo que enfatizan las etapas de modelación y diseño de las pruebas, tanto funcionales como unitarias, para facilitar la integración de los

modelos para reducción de casos de pruebas unitarias y funcionales con las propuestas establecidas en el mercado, a través de herramientas comerciales que en algunos casos se insertan en los ambientes de trabajo de los desarrolladores y probadores para la ejecución de pruebas tanto unitarias como funcionales. La contribución fundamental de MTest.search está en la propuesta de modelos de optimización que permiten diseñar casos de pruebas con combinaciones de valores reducidos y que tienen un soporte automatizado que puede ser fácilmente integrado a ambientes de trabajo. En este trabajo sólo se aborda lo relacionado con las pruebas funcionales.

En las Figuras 1 y 2 se muestran las actividades a seguir en la etapa de Captura de Requisitos para el diseño de pruebas funcionales.

Como parte de MTest.search se define un modelo de optimización que maximiza la cobertura de los escenarios en el proceso generación de combinaciones de valores de pruebas funcionales. Esta propuesta parte de la base de considerar que en la medida en que se prueben combinaciones de valores que cubran combinaciones de clases de equivalencia diferentes, se estará maximizando la cobertura de los escenarios. Por tanto, se propone un modelo de optimización, que maximiza la cobertura de los escenarios, maximizando las combinaciones de clases de equivalencias representadas en el conjunto de combinaciones de valores que se pretende obtener. Se trata, a su vez, de minimizar la cantidad de casos de prueba generados.

A continuación, se describe el modelo que será objeto de automatización:

Sea $\bar{\alpha} = (X_1, X_2, \dots, X_n)$ el vector que contiene las variables o atributos de entrada a la funcionalidad que se desea probar, $\bar{\beta} = (y_1, y_2, \dots, y_n)$ el vector que contiene la descripción del dominio de cada atributo perteneciente a $\bar{\alpha}$, C el nivel de cobertura de las combinaciones de valores que se desea lograr y n la cantidad de variables o atributos de entrada a la funcionalidad a la que se quiere generar valores de prueba.

Se define:

$$MO\left(\left(\bar{\alpha}, \bar{\beta}, C, f_o(\bar{\varepsilon}, \bar{\gamma})\right)\right) \rightarrow M_{lxn} \quad (1)$$

como el modelo de optimización por el cual se obtiene la matriz M_{lxn} con las combinaciones de valores de prueba posibles a partir de los criterios definidos en $\bar{\varepsilon}$ para los atributos descritos en $\bar{\alpha}$ y $\bar{\beta}$, que garantiza el nivel de cobertura de las combinaciones de valores de prueba especificado en C, donde:

l: es el número de combinaciones a obtener en la generación de combinaciones de valores de prueba y que es una función de la cobertura de los valores especificado como C.

M_{lxn}: es la matriz que contiene las combinaciones de valores generados y donde cada fila j se corresponde con un vector $\bar{\gamma}$, con $1 \leq j \leq l$. Por tanto, cada elemento w_{ji} se corresponde con un valor del atributo i en la combinación de valores j.

$\bar{\varepsilon} = (e_1, e_2, \dots, e_k)$: es el vector que contiene las k transformaciones a emplear para hacer discretos los dominios descritos en $\bar{\beta}$, que están sustentadas en técnicas de diseño de valores de casos de prueba para cada dominio diferente. Es decir, cada dominio tiene un vector $\bar{\varepsilon} = (e_1, e_2, \dots, e_k)$ diferente.

$\bar{\gamma} = (\varphi_1, \varphi_2, \dots, \varphi_n)$: es el vector que contiene una combinación de valores generada, en la que cada elemento φ_i es una triada ordenada $\varphi_i = (V_i, E_i, Z_i)$ donde E_i es el valor generado para el atributo i, V_i es el valor de la clase de equivalencia correspondiente y Z_i contiene el valor de verdad (1 si pertenece, 0 si no pertenece) correspondiente a la pertenencia de V_i al dominio y_i definido para el atributo X_i. Este vector constituye la codificación del problema de optimización. Cada elemento φ_i se obtiene a partir de la función de transformación:

$$T_{ik}(e_k, y_i) \rightarrow \bar{\sigma}_{ik} = (\lambda_1, \lambda_2, \dots, \lambda_n), \quad (2)$$

que aplica la transformación e_k al dominio $y_i \epsilon \bar{\beta}$ y obtiene l triadas ordenadas de la forma (V_i, E_i, Z_i).

A partir de esta transformación para un dominio de entrada se obtiene un conjunto discreto de valores basado en los criterios de diseño de casos de prueba:

$$f_o(\bar{\varepsilon}, \bar{\gamma}) = Max \sum_{j=1}^{l} fh(\bar{\gamma}_j) \quad (2)$$

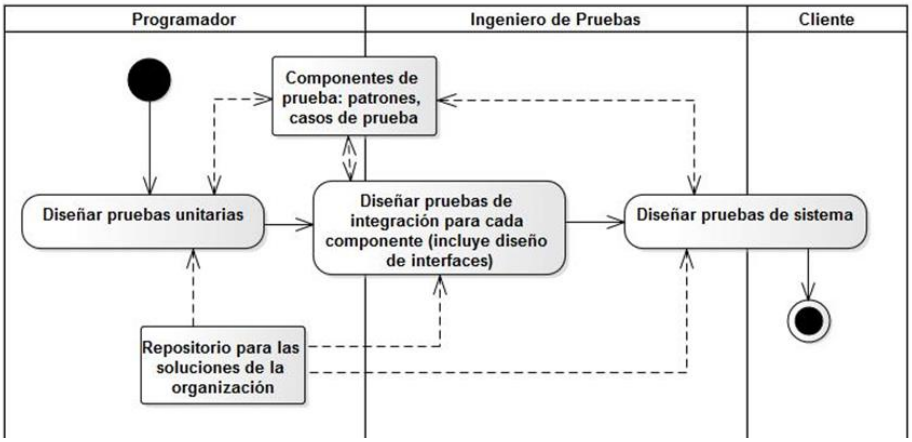

Fig. 1. Flujo para el diseño de pruebas funcionales

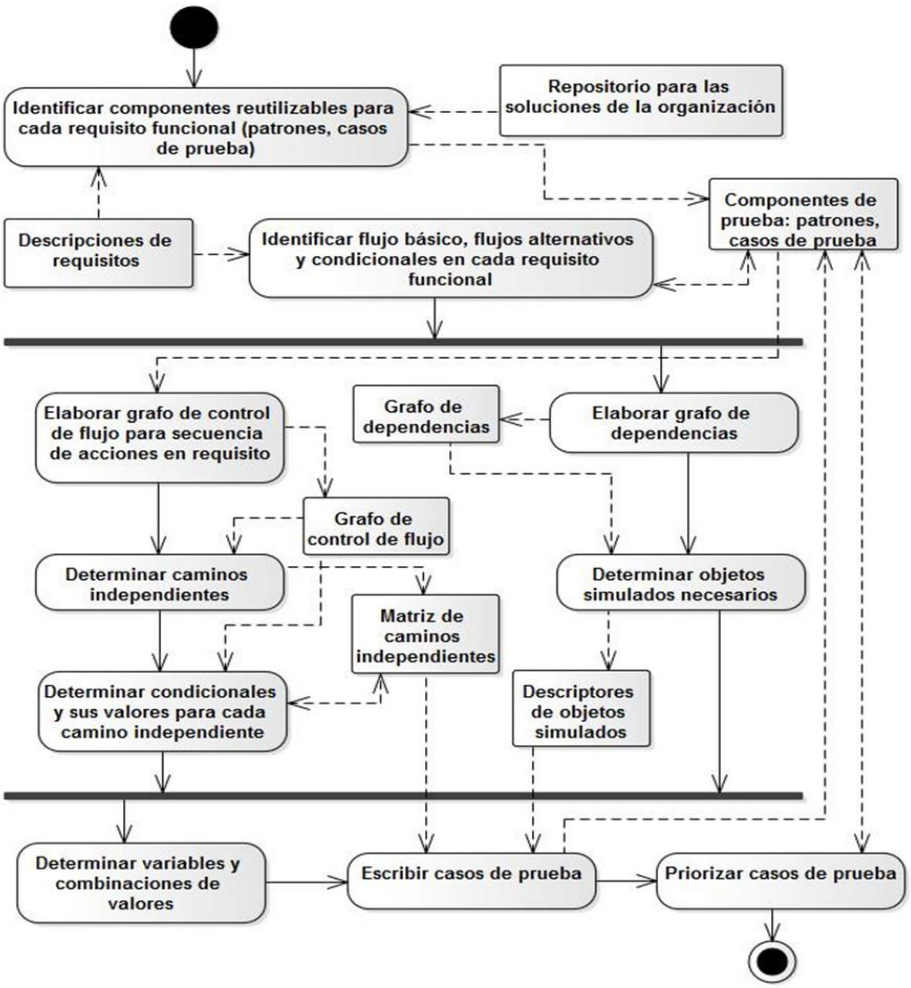

Fig. 2. Flujo para diseñar casos de pruebas para requisitos funcionales

es la función objetivo, que maximiza la cobertura de los escenarios para un nivel de cobertura de las combinaciones de valores especificado en C. $fh(\bar{y}_j)$: es la función heurística para cada una de las combinaciones de valores generadas. Esta considera la cantidad total de combinaciones de clases de equivalencia y la cantidad de filas de la matriz resultante que contienen la misma combinación de clases de equivalencia que el nuevo estado generado. Esto evita que se generen valores que cubran los mismos escenarios.

Es importante destacar que a diferencia de otras soluciones presentes en la bibliografía consultada, en las que el algoritmo propuesto para generar las combinaciones de valores, parte de rangos especificados para el dominio de cada una de las variables, la propuesta que se presenta en este modelo reduce los dominios de entrada a valores discretos utilizando el vector de transformaciones $\bar{\varepsilon} = (e_1, e_2, \dots, e_k)$. De esta forma el conjunto de valores de entrada se reduce a los valores más significativos para la prueba y a partir de aquí se aplica el algoritmo que ejecuta los elementos planteados en el modelo de optimización presentado anteriormente. Con esta modificación se limita el número de valores a probar sin disminuir la efectividad del diseño de las pruebas. Para la determinación de los vectores de transformación para cada tipo de datos se utiliza la técnica de diseño de casos de pruebas de particiones equivalentes.

A continuación se detallan las $T_{ik}(e_k, y_i)$ transformaciones para reducir cada dominio y_i a valores discretos aplicando como técnica de generación de valores: $e_k =$ Particiones_Equivalentes.

Para cada $y_i = "numérico"$ se obtienen los valores:

$$\lambda_1 = (V_i, 11,1),$$
$$V_i = \frac{MaxIntervalo(y_i) - MinIntervalo(y_i)}{2}$$
$$\lambda_2 = (V_i, 12,0),$$
$$V_i = add(MinIntervalo(y_i), -1)$$
$$\lambda_3 = (V_i, 12,0),$$
$$V_i = add(MaxIntervalo(y_i), 1)$$
$$\lambda_4 = (V_i, 13, pertenencia(y_i, V_i)),$$
$$V_i = decimal(MinIntervalo(y_i), MaxIntervalo(y_i))$$
$$\lambda_5 = (V_i, 14, aceptadoVacio(y_i, V_i)),$$
$$V_i = vacio(y_i)$$
$$\lambda_6 = (V_i, 15,1),$$
$$V_i = MinIntervalo(y_i)$$
$$\lambda_7 = (V_i, 15,1),$$
$$V_i = MaxIntervalo(y_i)$$

Para cada $y_i = "cadena"$ se obtienen los valores:

$$\lambda_1 = (V_i, 21,0),$$
$$V_i = generarCadena(add(longitud(y_i), 1))$$
$$\lambda_2 = (V_i, 22, aceptadoVacio(y_i, V_i)),$$
$$V_i = vacio(y_i)$$
$$\lambda_3 = (V_i, 23, pertenencia(y_i, V_i)) \ o$$
$$\lambda_3 = (V_i, 24, pertenencia(y_i, V_i)),$$
$$V_i = generarCadenaNumeros(longitud(y_i))$$
$$\lambda_4 = (V_i, 23, pertenencia(y_i, V_i)) \ o$$
$$\lambda_4 = (V_i, 24, pertenencia(y_i, V_i)),$$
$$V_i = generarCadenaSimbolos(longitud(y_i))$$
$$\lambda_5 = (V_i, 23, pertenencia(y_i, V_i)) \ o$$
$$\lambda_5 = (V_i, 24, pertenencia(y_i, V_i)),$$
$$V_i = generarCadenaLetras(longitud(y_i))$$
$$\lambda_6 = (V_i, 23, pertenencia(y_i, V_i)) \ o$$
$$\lambda_6 = (V_i, 24, pertenencia(y_i, V_i)),$$
$$V_i = generarCadenaNumerosSimbolos(longitud(y_i))$$
$$\lambda_7 = (V_i, 23, pertenencia(y_i, V_i)) \ o$$
$$\lambda_7 = (V_i, 24, pertenencia(y_i, V_i)),$$
$$V_i = generarCadenaNumerosLetras(longitud(y_i))$$
$$\lambda_8 = (V_i, 23, pertenencia(y_i, V_i)) \ o$$
$$\lambda_8 = (V_i, 24, pertenencia(y_i, V_i)),$$
$$V_i = generarCadenaSimbolosLetras(longitud(y_i))$$
$$\lambda_9 = (V_i, 23, pertenencia(y_i, V_i)) \ o$$
$$\lambda_9 = (V_i, 24, pertenencia(y_i, V_i)), V_i = generarCadenaSimbolosLetrasNumeros(longitud(y_i))$$

Para cada $y_i = "enumerado"$ se obtienen los valores:

$$\lambda_1 = (V_i, 31, aceptadoVacio(y_i, V_i)),$$
$$V_i = vacio(y_i)$$
$$\lambda_2 = (V_i, 32,1),$$
$$V_i = SeleccionarValor(y_i)$$
$$\lambda_3 = (V_i, 33,0),$$
$$V_i = GenerarNoPertenece(y_i)$$

ISSN 1405-5546
doi: 10.13053/CyS-21-3-2716

Para cada $y_i = \text{"}lógico\text{"}$ se obtienen los valores:

$\lambda_1 = (V_i, 41, aceptadoVacio(y_i, V_i)),$
$V_i = vacio(y_i)$
$\lambda_2 = (V_i, 42,1),$
$V_i = GenerarValorLogico(y_i)$
$\lambda_3 = (V_i, 43,0),$

$$V_i = GenerarNoPertenece(y_i)$$

Para cada $y_i = \text{"}fecha\_hora\text{"}$ se obtienen los valores:

$\lambda_1 = (V_i, 51,1),$
$V_i = GenerarFechaValida(y_i)\ o$
$V_i = GenerarHoraValida(y_i)$
$\lambda_2 = (V_i, 52,0),$
$V_i = GenerarFechaNoValida(y_i)\ o$
$V_i = GenerarHoraNoValida(y_i)$
$\lambda_3 = (V_i, 53, aceptadoVacio(y_i, V_i)),$
$V_i = vacio(y_i)$

Para cada $y_i = \text{"}conjunto\text{"}$ se obtienen los valores:

$\lambda_1 = (V_i, 61,1),$
$V_i = generarConjunto(longitud(y_i),$
$tipoElemento(y_i))$
$\lambda_2 = (V_i, 62,1),$
$V_i = generarConjunto(0, tipoElemento(y_i))$
$\lambda_3 = (V_i, 63,1),$
$V_i = generarConjunto(\frac{longitud(y_i)}{2},$
$tipoElemento(y_i))$
$\lambda_4 = (V_i, 64,0),$
$V_i = generarConjunto(add(longitud(y_i), 1),$
$tipoElemento(y_i))$
$\lambda_5 = (V_i, 65,0),$
$V_i = generarConjunto(\frac{longitud(y_i)}{2},$
$tipoElementoDistinto(y_i))$

Adicionalmente, en cada vector de transformaciones pueden ser incluidos valores que responden a la dependencia entre atributos Para ello se debe indicar si cada atributo es dependiente de otro. En el caso de los atributos dependientes estos son tratados como casos especiales en los que es necesario definir la relación de dependencia, que puede ser de: estructura, relatividad (mayor que, menor que, igual que) o existencia (pertenece o no a los elementos de un conjunto). Con esta información el vector de transformaciones del atributo dependiente es enriquecido.

3. Propuesta de solución

En el mercado existen diversas herramientas para la ejecución automática de pruebas, una vez que los desarrolladores o probadores han diseñado adecuadamente los casos de prueba. Por tal motivo, la propuesta de este modelo es que para dar soporte al modelo se utilice una combinación entre las herramientas de ejecución de pruebas existentes en el mercado y las herramientas que automatizan la generación de los casos de prueba, de forma tal que estas últimas se inserten en el entorno productivo de la empresa y se vinculen con las anteriores.

En el caso de las herramientas de generación de casos de prueba se ha implementado un componente que a partir de un Grafo de Control de Flujo obtiene los casos de pruebas. Este componente puede ser integrado a entornos de desarrollo productivo tanto para ejecutar pruebas funcionales como para ejecutar pruebas unitarias. En la Fig. 3 se muestra la relación de las aplicaciones propias del entorno productivo para la ejecución de pruebas con los componentes desarrollados como parte de este modelo, GeCaP, GeVaF [17] y GeVaU [10]. A la derecha aparecen las herramientas para pruebas funcionales y a la izquierda para pruebas unitarias.

Para el caso de las pruebas funcionales es necesario desarrollar aplicaciones clientes que se integren con el entorno de gestión del proyecto y de gestión de pruebas funcionales que utilice la empresa. Estas aplicaciones deben obtener la información de la funcionalidad que se pretende probar y una vez generados los escenarios y sus casos de prueba lo inserten en el entorno de pruebas funcionales.

El componente de generación automática de casos de pruebas (GeCaP) integra la generación de casos de pruebas unitarias y funcionales, para lo cual contiene componentes de generación de combinaciones de valores tanto para pruebas funcionales como para pruebas unitarias que

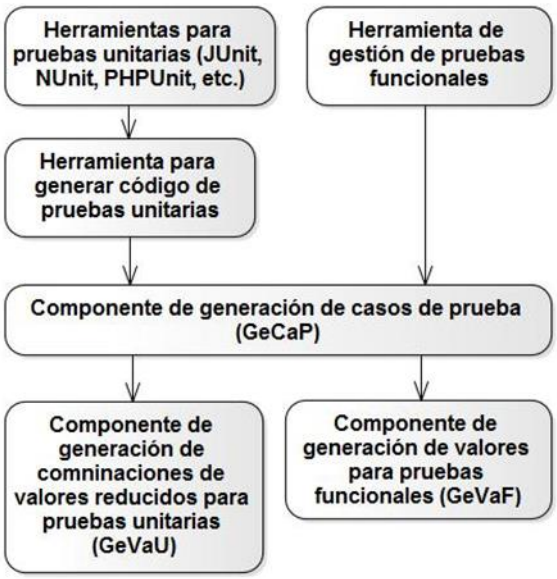

Fig. 3. Relación entre componentes del modelo y herramientas existentes en entorno productivo para ejecución de pruebas unitarias y funcionales

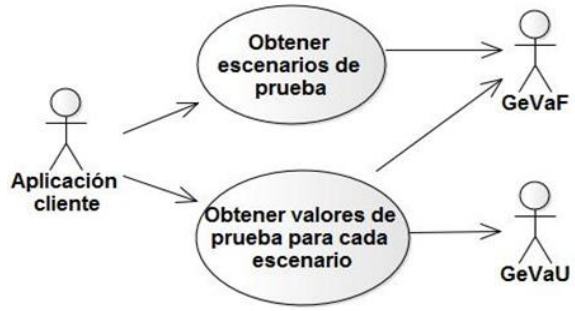

Fig. 4. Diagrama de casos de uso con las funcionalidades del componente GeCaP

utilizan metaheurística para optimizar el número de combinaciones de valores a generar.

En la Fig. 4 se muestran las funcionalidades del componente y su relación con los componentes que generan las combinaciones de valores para pruebas funcionales y unitarias, GeVaF y GeVaU respectivamente.

El componente GeVaF implementa el modelo de optimización, anteriormente definido, para la reducción de casos de pruebas funcionales haciendo uso de algoritmos heurísticos que maximizan la cobertura de los escenarios.

Para poder incorporar al entorno productivo la generación de los casos de pruebas funcionales utilizando el componente GeCaP se hace necesario desarrollar una aplicación de interfaz que capture los proyectos de desarrollo de la empresa, sus funcionalidades y los tipos de datos de cada una, y con esta información haga uso del componente. Se debe suministrar al componente el nivel de cobertura de los escenarios que se pretende alcanzar y la descripción de los atributos de entrada a cada funcionalidad.

Como resultado final el componente GeCaP genera un fichero texto con los valores de las variables para cada escenario. El fichero generado lo utiliza la aplicación cliente insertada en el entorno de gestión de pruebas funcionales.

4. Resultados y discusión

Con el objetivo de evaluar la viabilidad de introducir la solución propuesta en un entorno específico se desarrolló una aplicación cliente para el diseño de pruebas funcionales. La herramienta implementada para el entorno productivo de una empresa concreta hace uso del componente GeCaP para generar los casos de pruebas funcionales y unitarias. Ellas fueron probadas para un conjunto de proyectos reales de forma que se pudiera verificar la posible extensibilidad del modelo y el componente de software al entorno productivo de la empresa.

La evaluación del componente y el modelo estuvo guiada por las siguientes interrogantes:

1. ¿Es posible cubrir el 100% de los escenarios con un mínimo de combinaciones de valores para cada escenario, sin que se generen combinaciones de valores que cubran el mismo escenario?
2. Si el porcentaje de cobertura es menor que 100, ¿se obtienen resultados similares a los de máxima cobertura sin que se generen combinaciones que satisfagan el mismo escenario?
3. Si los niveles de cobertura de entrada son superiores al 100%, ¿se garantiza un cubrimiento similar de los escenarios y se minimiza la cantidad de combinaciones de valores repetidos?

Tabla 1. Descripción de las funcionalidades

Código de la funcionalidad	Número de atributos	Número de escenarios	Número de combinaciones de valores
01	2	81	9
02	3	27	27
03	2	27	12
04	4	504	108
05	2	28	9
06	3	112	27

Se diseñaron 3 experimentos con diferentes niveles de cobertura de escenarios para dar respuesta a las interrogantes planteadas, en los que se utilizó el método de búsqueda aleatoria. Como entrada, además de la cobertura, se suministran los atributos de cada funcionalidad y una descripción de sus dominios. A partir de esta información de entrada, los dominios de cada atributo se reducen a un conjunto de valores significativos para la prueba, según se explicó en las secciones correspondientes a los modelos de optimización a través de aplicar el vector de transformaciones.

Para el primer experimento se fijó el número de iteraciones de los algoritmos metaheurísticos de acuerdo con el 100% de cobertura de los escenarios y el segundo para el 80% de cobertura de los escenarios. Por último, se hace una corrida para un 120% de cobertura.

Se utilizó como caso de estudio una aplicación con 6 funcionalidades que como promedio tiene 3 atributos por cada una ellas. Las variables incluidas cubren los tipos de datos: cadena, numérico, enumerado, lógico y fecha.

En la Tabla 2, se muestran las características de las funcionalidades con respecto a la cantidad de atributos de cada una, la cantidad máxima de escenarios sin redundancia y la cantidad máxima de combinaciones de valores que pueden generarse.

La figura 5 muestra el gráfico con el resultado de las ejecuciones del modelo de optimización para las seis funcionalidades de la tabla 1 en correspondencia con el 120%, 100% y 80% de cobertura de escenarios, en la que se puede apreciar que la curva de generación de combinaciones de valores sigue igual distribución que la curva de cantidad mínima de combinaciones necesarias para garantizar el 100% de cobertura de escenarios y que para valores altos de escenarios se aprecia una mayor reducción de combinaciones de valores en la generación final.

Adicionalmente se presenta en la figura 6 un gráfico con la cantidad de combinaciones redundantes generadas en el peor caso para diferentes niveles de cobertura de escenarios. Este valor se obtiene para la funcionalidad 4 y 120% de cobertura de escenarios, en ese caso el 16,9% de las combinaciones generadas son redundantes. De forma general queda claro que hasta el 100% de cobertura de escenarios no se generan valores redundantes, por lo que la cantidad de combinaciones generadas es la mínima posible para el nivel de cobertura correspondiente.

A partir del análisis de los resultados se pudo comprobar que:

- Se garantiza cubrir los porcentajes de cobertura de los escenarios indicados en cada caso.
- Se generan cantidad de combinaciones de valores mínimas, en los casos de porcentajes de cobertura de hasta 100%, en particular se genera una única combinación para cada escenario.
- Para los porcentajes de cobertura superiores a 100% se generan combinaciones que satisfacen un mismo escenario, pero ningún escenario queda sin al menos una combinación de valores.

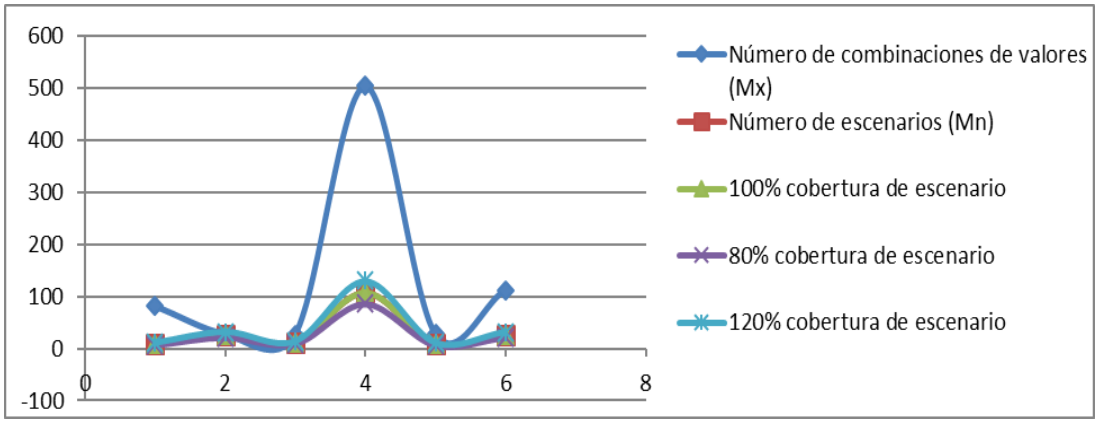

Fig. 5. Gráfico de dispersión de generación de combinaciones de valores para diferentes funcionalidades

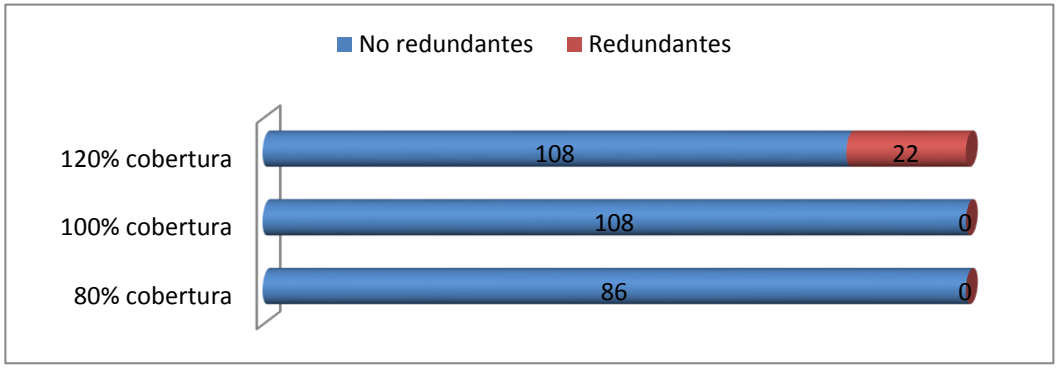

Fig. 6. Gráfico de comportamiento de la generación de combinaciones redundantes

5. Conclusiones

El presente trabajo facilita el proceso de pruebas funcionales durante el desarrollo de productos de software, debido a que permite automatizar el diseño de casos de pruebas funcionales. Además, permite reducir el tiempo y esfuerzo dedicado por los probadores y diseñadores de casos de prueba en el diseño y ejecución de pruebas funcionales.

Debido a que los valores de prueba generados tienen en cuenta las técnicas de particiones equivalentes y valores límites, las combinaciones de valores generados satisfacen todos los escenarios generando una cantidad de combinaciones de valores reducida y se garantiza la obtención de los valores para la cobertura exigida en cada caso.

Referencias

1. **Ahmed, B.S. & Zamli, K.Z. (2011).** Comparison of metahuristic test generation strategies based on interaction elements coverage criterion. *IEEE Symposium on Industrial Electronics and Applications (ISIEA),* Langkawi, Malaysia. DOI: 10.1109/ISIEA.2011.6108773.
2. **Anand, S., Burke, E.K., Chenc, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., & McMinn, P. (2013).** An orchestrated survey of methodologies for automated software test case generation. *The Journal of Systems and Software,* Vol. 86, No. 8, pp. 1978–2001. DOI: 10.1016/j.jss.2013.02.061.
3. **Bouquet, F., Grandpierre, C., Legeard, B., & Peureux, F. (2008).** A test generation solution to automate software testing. *Proc. of the 3rd International Workshop on Automation of Software*

Test, Leipzig, Germany. pp. 45–48, DOI:10.1145/1370042.1370052.

4. **Bregieiro, J.C. (2008).** 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,* Atlanta, USA. pp. 1819–1822, DOI:10.1145/1388969.1388979.
5. **Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., & Blackburn, M. (2014).** NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming,* Vol. 95, No. 3, pp. 275–297. DOI:10.1016/j.scico.2014.06.007.
6. **Chen, T., Zhang, X.S., Guo, S.Z., Li, H.Y., & Wu, Y. (2013).** State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems,* Vol. 29, No. 7, pp. 1758–1773. DOI: 10.1016/j.future.2012.02.006
7. **Díaz, E., Tuya, J., Blanco, R., & Dolado, J.J. (2008).** A tabu search algorithm for structural software testing. *Computers & Operations Research,* Vol. 35, No. 10, pp. 3052–3072. DOI: 10.1016/j.cor.2007.01.009.
8. **Elberzhager, F., Rosbach, A., Münch, J., & Eschbach, R. (2012).** Reducing test effort: A systematic mapping study on existing approaches. *Information and Software Technology,* Vol. 54, No. 10, pp. 1092–1106. DOI:10.1016/J.INFSOF.2012.04.007.
9. **Ferguson, R. & Korel, B. (1996).** The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM),* Vol. 5, No. 1, pp. 63–86.
10. **Fernández, P.B., Cantillo, W., Delgado, M.D., Rosete, A., & Yáñez, C. (2016).** Generación de combinaciones de valores de pruebas utilizando metaheurística. *Ingeniería Industrial,* Vol. 37, No. 2, pp. 200–207.
11. **Harman, M. (2007).** Automated test data generation using search based software engineering. *AST '07 Proceedings of the Second International Workshop on Automation of Software Test,* Minneapolis, USA. DOI: 10.1109/AST.2007.4.
12. **Harman, M., Mansouri, S.A., & Zhang, Y. (2012).** Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR),* Vol. 45, No. 1, DOI:10.1145/2379776.2379787.
13. **Hermadi, I., Lokan, C., & Sarker, R. (2014).** Dynamic stopping criteria for search-based test data generation for path testing. *Information and Software Technology,* Vol. 56, No. 4, pp. 395–407. DOI: 10.1016/j.infsof.2014.01.001.
14. **Iqbal, M.Z., Arcuri, A., & Briand, L. (2012).** 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,* Minneapolis, USA. DOI:11.45/2338965.2336777.
15. **Lamancha, B.P. & Polo, M. (2009).** Generación automática de casos de prueba para Líneas de Producto de Software. *Innovación, Calidad e Ingeniería del Software,* Vol. 5, No. 2.
16. **Lanzarini, L.C. & Battaiotto, P.E. (2010).** Dynamic generation of test cases with metaheuristics. *Journal of Computer Science & Technology,* Vol. 10, No. 2, pp. 91–96.
17. **Macias, A., Delgado, M.D., Fajardo, J., & Larrosa, D. (2016).** Generador de valores de casos de prueba funcionales. *Lámpsakos,* No. 15, pp. 51–58. DOI: 10.21501/21454086.1767.
18. **Memon, A.M., Pollack, M.E., & Soffa, M.L. (2001).** Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering.* Vol.27, No. 2, pp. 144–155. DOI: 10.1109/32.908959.
19. **Michael, C.C., McGraw, G., & Schatz, M.A. (2001).** Generating software test data by evolution. *IEEE Transactions on Software Engineering,* Vol. 27, No. 12, pp.1085–1110. DOI: 10.1109/32.988709.
20. **Myers, G.J., Badgett, T., & Sandler, C. (2011).** *The art of software testing*. 3rd Ed., New Jersey, USA: John Wiley & Sons.
21. **Pachauri, A. & Srivastava, G. (2013).** Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism. *The Journal of Systems and Software,* Vol. 86, No. 5, pp. 1191–1208, DOI:10.1016/j.jss.2012.11.045.
22. **Pargas, R.P., Harrold, M.J., & Peck, R.R. (1999).** Test-data generation using genetic algorithms. *The Software Testing Verification and Reliability,* Vol. 9, No. 4, pp. 263–282.
23. **Pressman, R.S. (2010).** *Software Engineering: A Practitioner's Approach*. 7th ed., New York, USA: McGraw-Hill.
24. **Ranjan, P., Mallikarjun, B., & Yang, X.S. (2013).** Optimal test sequence generation using firefly algorithm. *Swarm and Evolutionary Computation,* Vol. 8, pp. 44–53. DOI:10.1016/J.SWEVO.2012.08.003.

25. **Rongfa, T. (2012).** Adaptive Software Test Management System Based on Software Agents. *Advanced Technology in Teaching - Proceedings of the 2009 3rd International Conference on Teaching and Computational Science (WTCS 2009),* Berlin: Springer Berlin Heidelberg, pp.1–9. DOI:10.1007/978-3-642-25437-6_1.
26. **Sakti, A., Guéhéneuc, Y.G., & Pesant, G. (2012).** *Boosting search based testing by using constraint based testing: Search Based Software Engineering.* Springer Berlin Heidelberg, pp. 213–227.
27. **Sekhara, S., Hari, M.L., Kiran, U., Ch, S., & Ranjan, P. (2012).** Automated Generation of Independent Paths and Test Suite Optimization Using Artificial Bee Colony. *Proc. Engineering,* Vol. 30, pp. 191–200. DOI:10.1016/j.proeng.2012.01.851.
28. **Varshney, S. & Mehrotra, M. (2013).** Search based software test data generation for structural testing: a perspective. *ACM SIGSOFT Software Engineering Notes,* Vol. 38, No. 4, pp. 1–282. DOI: 10.1145/2492248.2492277.
29. **Wegener, J., Baresel, A., & Sthamer, H. (2001).** Evolutionary test environment for automatic structural testing. *Information and Software Technology,* Vol. 43, No.14, pp. 841–854. DOI: 10.1016/S0950-5849(01)00190-2.
30. **Ying, X., Yun-Zhan, G., Ya-Wen, W., & Xu-Zhou, Z. (2014).** Intelligent test case generation based on branch and bound. *The Journal of China Universities of Posts and Telecommunications,* Vol. 21, No. 2, pp. 91–97. DOI: 10.1016/S1005-8885(14)60291-0.
31. **Zhang, Z., Yan, J., Zhao, Y., & Zhang, J. (2014).** Generating combinatorial test suite using combinatorial optimization. *The Journal of Systems and Software,* Vol. 98, pp. 191–207. DOI: 10.1016/j.jss.2014.09.001.

Artículo recibido el 18/05/2017; aceptado el 29/05/2017.
Autor de correspondencia es Martha Dunia Delgado Dapena.