Complemento de VS.NET para la definición óptima de pruebas de *software* de caja negra mediante arreglos de cobertura*

Jaime Herney Meneses Ruiz[**]
Eduar Alexis Peña Velasco[***]
Carlos Alberto Cobos Lozada[****]
Jimena Adriana Timaná Peña[*****]
José Torres-Jiménez[******]

Recibido: 31/03/2016 • Aceptado: 16/03/2018
https://doi.org/10.22395/rium.v17n33a6

Resumen

Las pruebas de *software* pueden llegar a superar el 50 % del costo total de un proyecto de *software*, motivo por el cual las empresas necesitan una alternativa que permita reducir su costo y el tiempo de su realización. Esta investigación propone el uso de unas estructuras combinatoriales conocidas como arreglos de cubrimiento (CA) y arreglos de cubrimiento mixtos (MCA), que garantizan la detección hasta del 100 % de los errores con una mínima cantidad de pruebas. Con este enfoque, se desarrolló un complemento de *Visual Studio.NET* para la optimización de las pruebas y se evaluó su uso con estudiantes de último semestre de Ingeniería de Sistemas y de una empresa de *software*. Los resultados obtenidos son prometedores y motivan al grupo de investigación a divulgar su trabajo en el nivel nacional. El proyecto de investigación fue financiado por la Vicerrectoría de Investigaciones de la Universidad del Cauca.

Palabras clave: casos de prueba; pruebas combinatorias; arreglos de cubrimiento; arreglos de cubrimiento mixtos; recocido simulado; complemento de .Net.

* Artículo derivado del proyecto investigación VRI-4192 financiado por la Vicerrectoría de Investigaciones de la Universidad del Cauca y ejecutado entre 2015 y 2016.

** Ingeniero de Sistemas. Universidad del Cauca. Grupo de I+D en Tecnologías de la Información (GTI). Sector Tulcán, edificio FIET, Oficina 422, Popayán, Cauca, Colombia. Teléfono: (+57) 3138689529. Correo electrónico: jmeneses7@unicauca.edu.co. Orcid: https://orcid.org/0000-0002-3010-3962

*** Ingeniero de Sistemas. Universidad del Cauca. Grupo de I+D en Tecnologías de la Información (GTI). Sector Tulcán, edificio FIET, Oficina 422, Popayán, Cauca, Colombia. Teléfono: (+57) 3113709132. Correo electrónico: alexisp@unicauca.edu.co. Orcid: https://orcid.org/0000-0003-0138-3376

**** Doctor en Ingeniería de Sistemas y Computación. Universidad del Cauca. Grupo de I+D en Tecnologías de la Información (GTI). Sector Tulcán, edificio FIET, Oficina 422, Popayán, Cauca, Colombia. Teléfono: (+57) 3007379062. Correo electrónico: ccobos@unicauca.edu.co. Orcid: https://orcid.org/0000-0002-6263-1911

***** Maestría en Computación. Universidad del Cauca. Grupo de I+D en Tecnologías de la Información (GTI). Sector Tulcán, edificio FIET, Oficina 444, Popayán, Cauca, Colombia. Teléfono: (+57) 3006553149. Correo electrónico: jtimana@unicauca.edu.co. Orcid: https://orcid.org/0000-0002-1587-534X

****** Doctor en Ciencias Computacionales. Cinvestav Tamaulipas. Parque Científico y Tecnológico Tecnotam km. 5.5 carretera Cd. Victoria-Soto La Marina C.P. 87130 Cd. Victoria, Tamps, México. Correo electrónico: jtj@cinvestav.mx. Orcid: https://orcid.org/0000-0002-5029-5340

VS.Net Add-on for Optimal Definition of Black Box Software Testing Using Covering Arrays

Abstract

Software testing can exceed 50% of the total cost of a software project, which is why companies need an alternative to reduce their cost and time of implementation. This research proposes the use of combinatorial structures known *as Covering Arrays* (CA) and *Mixed Covering Arrays* (MCA), which guarantee the detection up to 100% of errors with a minimum amount of testing. With this approach, a Visual Studio.NET add-on was developed for the optimization of the tests and its use was evaluated with students of last academic term of Computer Science Engineering and of a software company. The results obtained are promising and motivate the research group to disseminate its work at the national level. The research project was funded by the Vice-Principal Office of Research of the Universidad del Cauca.

Keywords: test cases; combinatorial tests; covering arrays; mixed covering arrays; simulated annealing; .Net complement.

Complemento de VS.Net para a definição ideal de testes de software de caixa-preta por meio de *covering arrays*

Resumo

Os testes de software podem superar 50% do custo total de um projeto de software, motivo pelo qual as empresas precisam de uma alternativa que permita reduzir seu custo e o tempo de sua realização. Esta pesquisa propõe o uso de algumas estruturas combinatórias conhecidas como *covering arrays (CA)* e *mixed covering arrays (MCA)*, que garantem a detecção de até 100% dos erros com uma quantidade mínima de testes. Com esse enfoque, desenvolveu-se um complemento de *Visual Studio. NET* para a otimização dos testes e avaliou-se seu uso com estudantes do último semestre de Engenharia de Sistemas e de uma empresa de software. Os resultados obtidos são prometedores e motivam o grupo de pesquisa a divulgar seu trabalho nacionalmente. O projeto de pesquisa foi financiado pela Vice-reitoria de Pesquisas da Universidad del Cauca.

Palavras-chave: casos de teste; provas combinatórias; covering arrays; mixed covering arrays; recozimento simulado; complemento de .Net.

INTRODUCCIÓN

Para asegurar un producto con calidad, las empresas de desarrollo de software deben realizar una serie de pruebas que garanticen su óptimo funcionamiento antes de entregar el software al usuario final. Existen dos estrategias básicas que pueden ser usadas para diseñar pruebas: pruebas de caja negra y pruebas de caja blanca [1]. Las primeras son las de interés de la presente investigación. Estas pruebas son seleccionadas a partir de las especificaciones funcionales del software [2], en las que se proporcionan las entradas al componente software que se va a probar, se realiza la ejecución del componente y se determinan si las salidas producidas son equivalentes a las esperadas [3, 4].

Las pruebas de software son una de las tareas fundamentales y más costosas para el control de la calidad del software. Por lo general estás pruebas se realizan bajo limitaciones de tiempo y presupuesto. Como se evidencia en [5, 6] el costo de realizar pruebas puede llegar a superar el 50 % del costo total del desarrollo del producto, motivo por el cual la comunidad científica ha enfocado sus esfuerzos en la investigación de técnicas que permitan optimizar este proceso [7]. Aunque los resultados de investigación se han empezado a adoptar de forma gradual en la industria del software, todavía existe una gran brecha entre los sistemas de prueba de software y la necesidad de las empresas por disminuir los costos y el tiempo de aplicación de los casos de prueba.

En términos generales, los casos de prueba se generan a partir de la información presente en algún artefacto como los modelos, documentos de especificación de requisitos, código fuente, estructura del programa o información obtenida en tiempo de ejecución [5]. En [8] se muestran diferentes algoritmos de búsqueda para la construcción de casos de prueba. Normalmente, los casos de prueba son generados empíricamente por un experto y sin usar un método formal, lo cual termina en pérdidas de tiempo y dinero [5].

Las pruebas de software exhaustivas permiten probar un componente en su totalidad. Sin embargo, es una tarea poco práctica, costosa y que consume demasiado tiempo ya que genera un número elevado de casos de prueba debido a las múltiples combinaciones a las que da lugar [9].

Como estrategia alternativa a las pruebas exhaustivas se encuentran las pruebas combinatorias, las cuales definen un conjunto de pruebas más pequeño, que es comparativamente más fácil de administrar y ejecutar y que, una vez identificadas correctamente, permiten tener una máxima cobertura en la prueba [10].

Uno de los objetos combinatorios que satisfacen este tipo de pruebas son los arreglos de cobertura (CA) y los arreglos de cobertura mixtos (MCA) [11]. Un CA es un objeto matemático útil para el desarrollo de pruebas funcionales que busca asegurar la inexistencia de errores en el software a través de la generación del menor número de casos de prueba que cubran todos los conjuntos de interacciones de los parámetros de entrada de un procedimiento, función o unidad lógica de software [12]. El uso de CA y MCA reduce costos y esfuerzos y aumenta significativamente la efectividad de las pruebas de software [13].

Por otro lado, la realidad de las empresas de desarrollo de software en Iberoamérica, muestra que la gran mayoría de ellas: i) en general están conformadas por equipos pequeños [14], ii) no cuentan con el personal suficiente como para hacer las pruebas de forma exhaustiva y iii) carecen de una herramienta automática para hacer la traducción de los datos de prueba reales a CA o MCA y viceversa y además, esta funcionalidad no está incluida en el entorno integrado de desarrollo (Integrated Development Environment, IDE) que usan en la construcción de sus aplicaciones. De allí surge la necesidad de disponer de un complemento software que se vincule a un IDE y que permita el uso de CA y MCA para el diseño de pruebas de caja negra, optimizando el proceso de pruebas de calidad en las empresas colombianas.

Por lo anterior, en esta investigación se presenta una alternativa de solución ante la necesidad de optimizar la tarea de generación de casos de prueba, realizándolas en menos tiempo, con menos esfuerzo y con la más alta probabilidad de detectar fallos que el software pueda presentar. Aunque actualmente existen programas como JUnit o NUnit, para facilitar la pruebas de software, ninguno utiliza los CA como base para crear los casos de prueba, ahí radica la diferencia de este trabajo con lo existente, puesto que al usar esta estructura se garantiza que cada dupla, tripleta, cuarteto, quinteto o sexteto (fuerza de interacción 2, 3, 4, 5 y 6 respectivamente) de parámetros se prueba por lo menos una vez, lo que garantiza la detección hasta del 100 % de los fallos según la evidencia empírica mostrada en [6, 15, 16].

A continuación, en la sección 1 se presenta un resumen del proceso desarrollado en la investigación. Luego, en la sección 2 se describe el complemento desarrollado y sus componentes, que incluyen el posoptimizador, la adaptación del algoritmo de recocido simulado y las principales interfaces. Después en la sección 3, se muestran los resultados obtenidos de la evaluación del complemento con pruebas alfa y beta, realizadas con estudiantes universitarios y *tester* de una empresa de software de Popayán (Colombia) respectivamente. Finalmente, en la sección 4 se presentan las conclusiones de la investigación y el trabajo futuro que el grupo de investigación espera desarrollar en el área.

1. PROCESO INVESTIGATIVO

Para el desarrollo de esta investigación se llevaron a cabo cuatro fases basadas en instancias del patrón de investigación iterativo (Iterative Research Pattern). En dicho patrón se tienen en cuenta cuatro etapas básicas que son: observación del problema, identificación del problema, construcción de la solución y evaluación de la solución.

En la primera fase, se trabajó en la definición de un algoritmo de posoptimización de CA o MCA y la comparación de sus resultados obtenidos con los reportados en el estado del arte. La importancia del algoritmo de posoptimización radica en que cuando se busca en un repositorio de arreglos de cobertura un CA o MCA específico es posible que este no se encuentre. En este caso se puede proceder de dos formas: la primera, consiste en ejecutar un algoritmo (normalmente metaheurístico, voraz, aproximado o exacto) que lo cree desde cero, lo cual puede involucrar un alto costo computacional y de tiempo; y la segunda, se puede tomar un CA o MCA del repositorio que lo contenga, es decir que cuente con más columnas (parámetros o factores) del que se está buscando, se le eliminan las columnas que no se requieren y finalmente, se le eliminan tantas filas como sea posible manteniendo las características del CA o MCA requerido. Esta segunda forma de proceder es menos costosa computacionalmente y en tiempo, aunque no siempre garantiza la construcción de un CA o MCA óptimo (menor número de filas posible), y resume las tareas principales de un algoritmo de posoptimización de CA o MCA.

En la segunda fase se trabajó en la adaptación de un algoritmo de recocido simulado del estado del arte para la creación fuera de línea de un CA o MCA específico desde cero. Buscando que se enriquezca el repositorio de CA o MCA, cada solicitud de posoptimización (no se encontró el CA o MCA específico y se tuvo que posoptimizar uno más grande del repositorio, lo que conllevó a entregar una respuesta rápida que posiblemente no es la óptima) se encola para luego, fuera de línea, poder encontrar con este algoritmo, los CA O MCA óptimos y usarlos en futuras solicitudes de los *testers*.

En la tercera fase se diseñó e implementó el complemento como tal usando desarrollo rápido de aplicaciones (DRA) como proceso de desarrollo. Esto incluyó la organización del repositorio, el servicio web, las interfaces de usuario y un codificador-decodificador que se encarga de traducir las necesidades del *tester* en un CA o MCA y viceversa.

Finalmente, en la cuarta fase se realizó la evaluación del complemento con una prueba alfa con estudiantes de Ingeniería de Sistemas de últimos semestres y dos pruebas beta con *testers* de una empresa de software de Popayán, y el correspondiente desarrollo de las mejoras al complemento con base en los comentarios recibidos en cada una de las pruebas. Al finalizar cada fase se realizó la documentación del proceso a la fecha y se planeó el desarrollo de la siguiente fase.

2. COMPLEMENTO DESARROLLADO

El complemento de Visual Studio.Net (VS.Net) se desarrolló por componentes: un algoritmo de posoptimización, la interfaz gráfica de usuario (GUI) que se usa dentro del VS.Net, un codificador-decodificador, un repositorio (base de datos SQL-Server) y un servicio web que permite la comunicación entre el complemento en el lado del cliente (VS.Net) y el repositorio en el lado del servidor. En la figura 1 se explica la dinámica de los componentes cuando un *tester* lo usa.

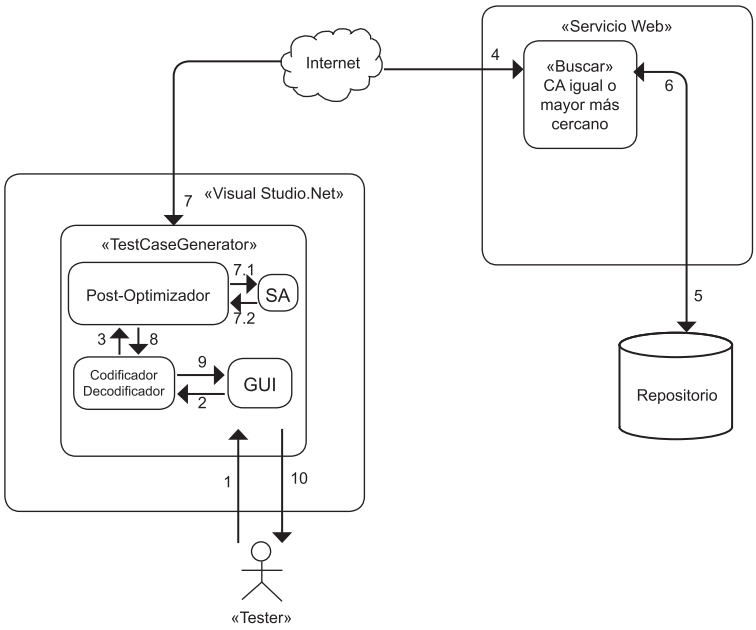

Figura 1. Esquema del prototipo para la generación de casos de prueba

Fuente: elaboración propia

El *tester* utiliza la interfaz gráfica de usuario (GUI) para cargar un archivo .EXE o .DLL (que contiene las clases con los métodos que se van a probar) y selecciona el método que desea probar, luego selecciona la fuerza de interacción (valor de t para el CA o MCA a usar, el cual determina el nivel de exhaustividad de los casos de prueba que se generarán) y configura los valores de entrada para cada variable (paso 1 de la figura 1). El complemento transforma las entradas usando el codificador-decodificador y a través de los otros componentes busca en el repositorio un CA o MCA acorde a esa necesidad, si lo encuentra lo recupera y lo utiliza, de lo contrario busca el más cercano (uno que lo subsuma) y lo adapta (paso 2 de la figura 1).

El codificador-decodificador solicita al posoptimizador, el CA o MCA apropiado a las necesidades del *tester* (paso 3), el posoptimizador solicita al servicio web dicho

CA o MCA (paso 4), el servicio web busca en el repositorio y entrega el más apropiado (paso 5). Cuando el posoptimizador recibe el CA o MCA (pasos 6 y 7) el optimizador pone a correr el algoritmo de recocido simulado (Simulated Annealing, SA) en el trasfondo (*background*) para encontrar un mejor CA o MCA para futuras pruebas (este paso es opcional y se ejecuta solo si es requerido, pasos 7.1 y 7.2), luego, se realiza el proceso de posoptimización si el CA o MCA recibido es más grande (subsume) que el verdaderamente requerido y entrega el resultado al codificador-decodificador (paso 8).

Cuando el posoptimizador reporta el CA o MCA requerido se transforma en casos de prueba mediante el uso del codificador-decodificador que reemplaza cada fila en un caso de prueba con los valores configurados y los muestra al *tester* (paso 9). El *tester* los visualiza, configura cada uno de los valores esperados que retorne el método y los guarda. Por último, el complemento genera el código fuente para copiar y pegar directamente sobre un proyecto de pruebas NUnit u otro (paso 10).

2.1 El Posoptimizador

El proceso de posoptimización tiene como objetivo reducir un CA de fuerza t y de alfabeto V1 hasta uno de fuerza t y alfabeto V2, donde V2 < V1. La reducción se realiza con los pasos presentados en la tabla 1**.**

Tabla 1. Descripción del método de posoptimización

1. Leer el CA de alfabeto V1 tomado del repositorio y ubicarlo en una matriz C de N*K, donde N y K son respectivamente, el número de filas y columnas del CA.
2. Pasar la matriz C al alfabeto requerido (V2), dejando en cada columna solo el valor máximo permitido (valores válidos en esa columna) y convirtiendo los números que no pertenecen al alfabeto (mayores a V2), en comodines (simbolizados con un guion).
3. Borrar las filas que tienen más de K - t comodines, es decir que no cumplen con la fuerza t.
4. Borrar las filas repetidas y ordenar la matriz según la cantidad de comodines, dejando de últimas las filas con mayor número de comodines.
5. Arrancando desde la última fila de la matriz C, repetir el siguiente proceso fila por fila mientras no se haya llegado hasta la primera fila de la matriz C y además se siga disminuyendo la cantidad de filas en la matriz C.
5.1. Crear o actualizar la lista de combinaciones repetidas.
5.2. Buscar comodines de acuerdo con la lista de combinaciones repetidas.
5.3. Borrar filas repetidas y ordenar por cantidad de comodines.
5.4. Tomar parejas de filas y representarlas en una sola de ser posible.

Fuente: elaboración propia.

2.2 El algoritmo de recocido simulado (SA)

El recocido simulado o *Simulated Annealing* (SA) es una técnica de optimización estocástica de propósito general que surge de una analogía con el proceso de calentamiento y enfriamiento de metales usado en la industria para obtener materiales más resistentes y con mejores cualidades [17].

En el año 2003, Cohen *et al.* [18] aplicaron el algoritmo de SA para la construcción de arreglos de cobertura. Los resultados de esta investigación evidenciaron que SA permite construir CA más pequeños que otros métodos empleados para el mismo fin. Hoy en día los CA y MCA generados a partir de SA y sus posteriores refinamientos y variaciones han proporcionado los resultados más precisos y competitivos a la fecha [19].

La implementación computacional de esta técnica requiere de tres parámetros: temperatura inicial T_0, temperatura final T_f y el grado de enfriamiento α ($0 < \alpha < 1$). Para iniciar con el proceso del recocido simulado, la temperatura T_f se establece como T_0, posteriormente en cada ciclo de repetición se asigna αT_t a cada iteración T_{t+1} hasta lograr el valor de T_f. Basado en la analogía de los cambios de estado del metal en el algoritmo se presentan como S_i con energía E_i y cada cambio de estado se produce por medio de perturbaciones, representados como S_j con energía E_j. Si S_j con energía E_j es menor que S_i con energía E_i, se establece ese nuevo estado, de lo contrario S_j es sometido a una probabilidad para poder ser aceptado ($\text{probabilidad} = e^{-\left(\frac{Ej-Ei}{Kb*Tt}\right)}$, donde *Kb* es la constante de Boltzmann). Por último, se define L como un cuarto parámetro, que define el número de perturbaciones realizadas sobre la temperatura T_t.

Para lograr que el algoritmo no quede en un ciclo infinito, este termina su ejecución si se encontró una solución deseada, si la temperatura final fue alcanzada o si la mejor solución (solución global) en el conjunto de soluciones no cambia [17].

En esta investigación se realizó una adecuación del método de recocido simulado propuesto en [20] junto con dos funciones de vecindad que permiten el hallazgo de nuevas y mejores soluciones para los CA. Las funciones utilizadas se denominan *NF1* y *NF2, d*onde *NF1* recoge una t-tupla faltante del CA al azar e intenta establecerla en la primera fila elegida de manera aleatoria en la matriz, permitiendo mejorar la solución actual y *NF2* selecciona al azar *2t* celdas de la matriz y evalúa todos los posibles cambios de símbolo de las celdas seleccionadas, estableciendo solo el primer cambio de símbolo que permite mejorar la solución actual y, dado el caso de la no existencia de tal cambio de símbolo, se aplica el cambio que genera el mejor vecino. La función de vecindad *NF2* también busca comodines en las celdas seleccionadas y establece un símbolo al azar en sus correspondientes celdas.

En el proceso de búsqueda, *NF1* se utilizó con una probabilidad p y *NF2* con una probabilidad *(1 – p).* Donde *p* varía entre 0,55 y 0,85. De acuerdo con lo expresado en [20], se determina que para un óptimo funcionamiento del recocido simulado, las variables se definen con un valor inicial de: a) $T_0 = 1$; b) $\alpha = 0{,}99$; c) $L = Nkv^2$, d) $T_f = 0{,}0000000001$; e) $FF = 33$; y f) $kmax = t + 1$. En la tabla 2 se presenta en pseudocódigo la adaptación del algoritmo de recocido simulado y las funciones de vecindad mencionadas.

Tabla 2. Adecuación del algoritmo de recocido simulado y funciones de vecindad

```
Función RecocidoSimulado: retorna booleano
Entradas: Matriz cRecibido, arreglo v, entero n, entero k, entero t, real probabilidad
Matriz Ca, C, C', C''
p ← probabilidad
T ← 0, T0 ← 1, Tf ← 0.0000000001, α ← 0.99
L ← nkv2, k'← 0
FC ← 0, FF← 33, Kmax ← t+1
Ca ← copiarC (cRecibido, n, k)
C ← copiarC (cRecibido, n, k)
evalC ← getTamanoFaltantes (), evaluarActual ← evalC+1
para i ← 1 hasta 2 Con paso 1 haga
        mientras (T > Tf y f (Ca, n, k, v, t) > 0 y FC < FF y evaluarActual != 0) haga
            para j ← 0 hasta L y evaluarActual != 0 Con paso 1 haga
                rand ← random ()
                si (rand < p) entonces C''← NF1 (C', n, k, v, t, k', evaluarActual)
                sino C''← NF2 (C', v, evaluarActual, n, k, t, k')
                min ← minimo (evaluarActual, evaluarC, temp) // aprendizaje de Boltzmann
                si (rand < min o evaluarActual < evaluarC) entonces
                        C ← copiarC (C'', n, k)
                        evalC ← evaluarActual
                        k' ← 0
                    sino
                        k'← k'+1
                    fin
                si (f (C, n, k, v, t) < f (Ca, n, k, v, t)) entonces
                        Ca ← copiarC (C, n, k)
                        FC ← 0
                fin si
                si (k' < kmax) entonces k'← 0
            fin para
            T ← T * α
            FC ← FC + 1
        fin mientras
fin para
cRecibido = Ca
si (TestCA (Ca, v, n, k, t)) entonces retorne verdadero
sino retorne falso
```

Fuente: elaboración propia

2.3 Interfaz gráfica de usuario (GUI)

El complemento desarrollado está disponible para Visual Studio.Net en sus versiones 2010, 2012 y 2013. La figura 2 muestra la interfaz donde el *tester* selecciona la clase y el método a probar, define además los parámetros y los valores que se tomarán como prueba y los botones de generación de pruebas. Además, permite gestionar las pruebas que se han desarrollado previamente.

La figura 3 muestra la interfaz que usa el *tester* para personalizar la prueba con el CA o MCA recibido. Allí define los valores esperados para cada caso de prueba y en la parte derecha puede seleccionar y copiar el código generado por el complemento.

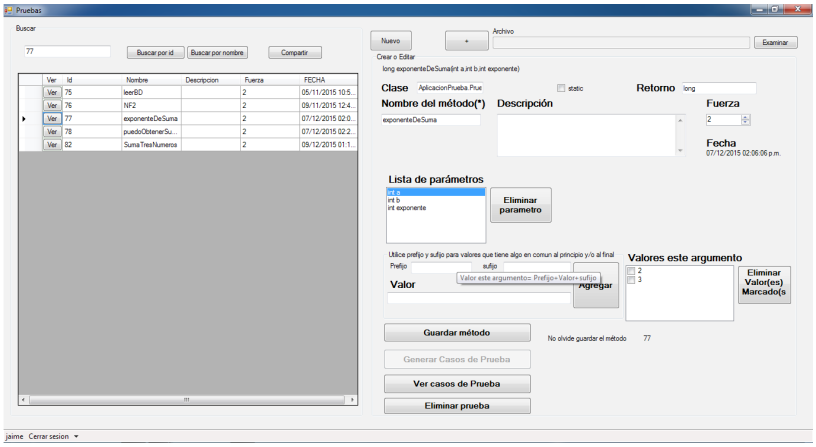

Figura 2. Interfaz principal para la configuración y gestión de pruebas

Fuente: elaboración propia

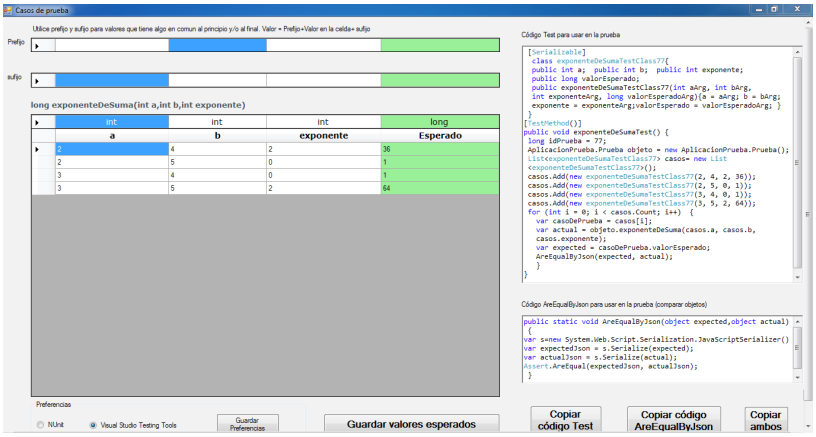

Figura 3. Interfaz para configurar parámetros de salida, valores esperados y copiar código fuente

Fuente: elaboración propia

3. RESULTADOS Y EXPERIMENTACIÓN

En esta sección se muestran los resultados obtenidos en la adaptación de los CA y MCA utilizados por el complemento, comparándolos con los existentes en el repositorio de CA (actualmente privado) del Cinvestav (Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional de México) en Tamaulipas. Luego, se muestran los resultados de la experimentación que se realizó con un conjunto de estudiantes de los últimos semestres del programa de Ingeniería de Sistemas en la Universidad del Cauca y el grupo Ideti, empresa perteneciente a la incubadora Cluster CreaTIC de Popayán.

3.1 Evaluación del posoptimizador

Teniendo en cuenta que el complemento es una aplicación que debe operar en tiempo real y requiere resultados rápidos, el repositorio se pobló inicialmente con CA y MCA con los alfabetos más comúnmente utilizados, a saber, entre 3 y 10 parámetros (k) y fuerza de interacción (t) entre 2 y 6.

En la tabla 3 se realiza una comparación de los MCA reportados por el Cinvestav y los adaptados por el método de recocido simulado y posoptimización propuestos en esta investigación. Se evidencia que, en promedio, los MCA reportados por el Cinvestav

Tabla 3. Comparación de resultados entre los MCA reportados en el Cinvestav y los adaptados con el algoritmo de posoptimización propuesto

Id	*MCA*	*t*	*Reportados Cinvestav*	*Adaptados*	*Diferencia*	*%*
1	3^3 2^3	2	9	9	0	0,00
2	6^1 5^5	2	30	34	4	13,33
3	6^4 5^2	2	37	45	8	21,62
4	6^5 5^1	2	39	46	7	17,95
5	5^1 3^4	3	45	45	0	0,00
6	4^1 3^4	2	12	12	0	0,00
7	4^1 2^8	2	8	8	0	0,00
8	4^3 3^5	2	16	16	0	0,00
9	4^5 3^4	2	19	19	0	0,00
10	4^6 3^3	2	20	20	0	0,00
11	4^6 3^4	2	20	22	2	10,00
12	5^1 4^5	2	20	20	0	0,00
	Total		275	296	21	7,63

Fuente: elaboración propia

son mejores en un 7,63 %. El hecho de tener unos resultados menos favorables se debe a que el objetivo de esta investigación (contrario al del Cinvestav) no consiste en conseguir el mejor CA o MCA, sino obtener un buen resultado en poco tiempo y con una máquina accesible para cualquier empresa. Además, la máquina involucrada para la creación de los CA y MCA del Cinvestav consta de un *cluster* con las siguientes características: 11032GB de RAM, 4244 cores CPU, disco duro de 54TB y sistema operativo Linux, mientras que la utilizada para la adaptación de los CA y MCA en la presente investigación fue un PC de escritorio con las siguientes especificaciones: 4GB de RAM, un disco duro de 1TB, un procesador Intel(R) Core(TM) i7-3770 CPU y un sistema operativo Windows 7 de 64bits.

3.2 Pruebas realizadas con estudiantes y el grupo Ideti

Con el propósito de poner a prueba el prototipo funcional del complemento, se realizó un experimento en dos salas de la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca con 22 estudiantes del programa de Ingeniería de Sistemas, en el marco de las pruebas alfa y dos pruebas en las instalaciones del Cluster CreaTIC (Popayán) con ocho ingenieros del grupo Ideti, en el marco de las pruebas beta.

El experimento de la prueba alfa se llevó a cabo de la siguiente manera:

- Se presentó una breve introducción de lo que se iba a realizar en la prueba y se entregaron todos los insumos.
- Los participantes realizaron una primera prueba utilizando un video como guía de aprendizaje para interiorizar todos los pasos.
- Se importó una clase desde un archivo DLL y se realizaron los casos de prueba para cada uno de los métodos.
- Se realizó una encuesta final a cada *tester,* con preguntas abiertas y cerradas.

La prueba evalúa los siguientes parámetros: facilidad de uso, tiempo para ingresar los parámetros de entrada (configurar prueba), tiempo de respuesta, cantidad de errores detectados sobre el ejecutable probado. Para proceder con la evaluación se realizó una encuesta para conocer la satisfacción de los participantes en los experimentos. En la tabla 4 se muestra cada una de las características evaluadas. En la prueba alfa con los estudiantes se evaluaron las primeras once características, luego con la inclusión de las mejoras se agregaron tres características más para completar las catorce de la tabla. Las características se evaluaron usando entre dos y cinco opciones posibles usando la semántica apropiada para cada característica, por ejemplo, para la característica 1 relacionada con el "Grado de satisfacción en la realización de la prueba", las

Tabla 4. Características evaluadas y nomenclatura de colores

N°	Pregunta
1	Grado de satisfacción
2	Comparación con otra herramienta
3	Aseguran volver a utilizar la herramienta
4	Recomendaría la herramienta
5	La herramienta cubre las necesidades
6	Fácil de usar
7	Complejidad
8	Rapidez en tiempo de respuesta
9	Tiempo usado en comparación con otra herramienta
10	Cantidad de errores detectados sobre el ejecutable
11	Tiempo usado para configurar la prueba
12	La nueva versión es
13	Es visible el funcionamiento del campo prefijo y sufijo
14	Son visibles los datos del método que se está probando

Nomenclatura de colores

Favorable	Intermedio	No sabe	Desfavorable

Fuente: elaboración propia

opciones fueron: completamente satisfecho, satisfecho, insatisfecho y completamente insatisfecho; mientras que la característica 3 que evalúa si el *tester* "Recomendaría la herramienta" cuenta solo con las opciones de sí y no.

En la figura 4 se observa que los resultados obtenidos en el experimento son favorables en los once aspectos en los que se enfocó la encuesta. Sin embargo, las barras color naranja evidencian que la herramienta no presenta algún cambio importante con respecto a otras herramientas y por último las barras rojas evidencian inconformidad por parte de los encuestados, esto se tomó como oportunidades de mejora. La información necesaria para conseguirlo se obtuvo mediante preguntas abiertas en las que el encuestado expresó libremente, lo que le gustó, lo que no le gustó y lo que cambiaría para mejorar. Lo cual se resumió en dos aspectos: i) cargar el ejecutable una vez y usarlo tantas veces como sea necesario, ii) permitir un prefijo para definir los valores de las variables y facilitar así la inclusión de los parámetros.

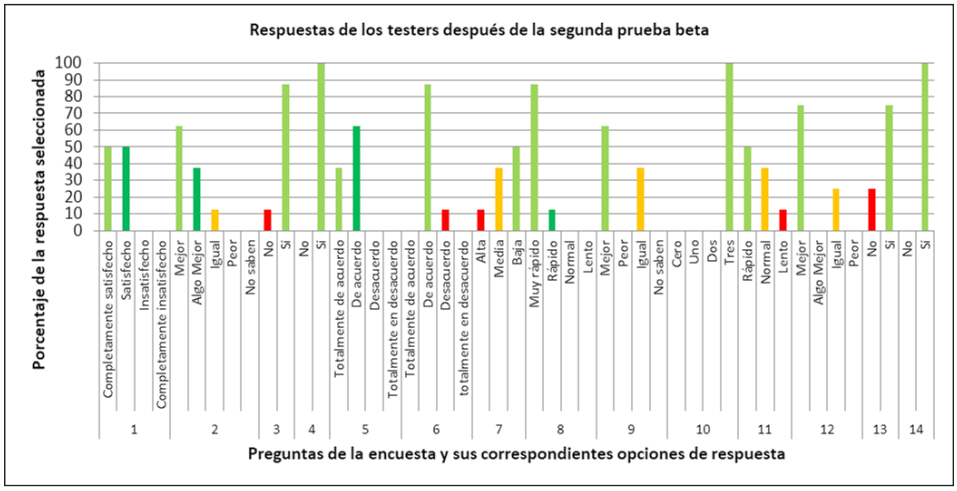

Figura 4. Gráfica que representa la información de la encuesta de los 22 estudiantes

Fuente: elaboración propia.

La primera prueba beta con el grupo Ideti implicó los mismos pasos del experimento de la prueba alfa con estudiantes y, adicionalmente, la interacción libre de los *testers* con el complemento. Como resultado, los *testers* solicitaron nuevas mejoras, a saber: i) ayudas y etiquetas a ciertos campos de la interfaz, ii) mejorar la forma como se configuran los valores esperados y parámetros de salida y iii) mostrar el tipo de error que se presenta, no solo que hay error.

En la figura 5 se muestran los resultados finales de la encuesta a los usuarios de Ideti luego de hacer las mejoras al complemento (ejecutadas ya las dos pruebas beta). Se puede observar que el complemento desarrollado obtuvo mejoría en su prototipo final, debido a que se corrigieron las falencias detectadas en el primer prototipo. Por otra parte, el grupo Ideti mostró conformidad con las pruebas realizadas a través del complemento desarrollado y aseguraron utilizarla en un futuro cercano en el desarrollo de sus proyectos software.

El único cambio que no se pudo desarrollar fue el relacionado con que el complemento muestre el tipo de error que se presenta, no solo que se presenta un error. Lo anterior debido a que dicho requisito desbordaba el alcance de la investigación y requiere otro tipo de técnicas, los CA y MCA en pruebas de caja negra no están diseñados para resolver este requisito.

4. CONCLUSIONES Y TRABAJO FUTURO

El algoritmo de posoptimización propuesto para adaptar los CA o MCA del repositorio a los requerimientos de las pruebas cumplió con el objetivo y lo hizo de forma rápida, usando una máquina accesible para cualquier empresa.

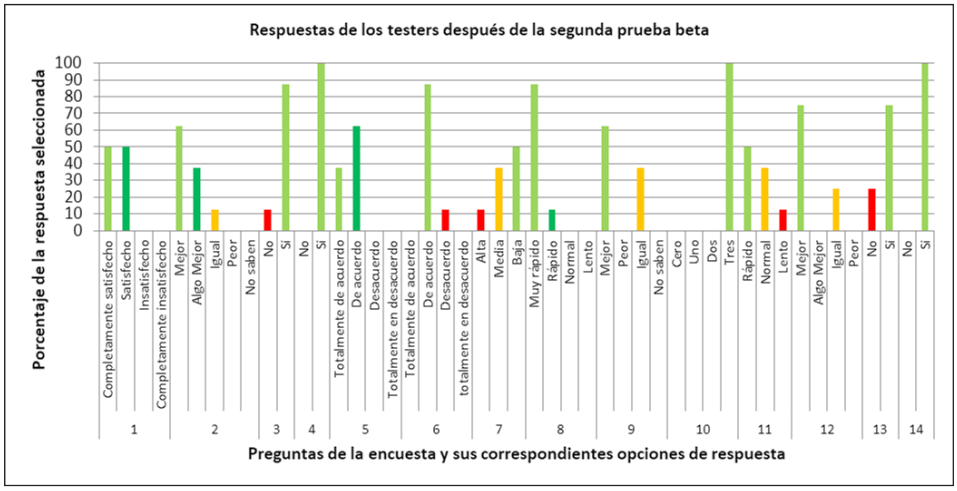

Figura 5. Gráfica que representa la información de la encuesta realizada al grupo Ideti al finalizar los ajustes sobre el complemento

Fuente: elaboración propia.

El grado de optimización del algoritmo propuesto en esta investigación es muy bueno, puesto que los CA y MCA obtenidos son muy similares a los mejores reportados por el Cinvestav, teniendo en promedio un 7,63 % de filas adicionales.

Se desarrolló un complemento (*plug-in*) para Visual Studio .Net capaz de soportar la generación de casos de prueba basados en arreglos de cobertura mediante el algoritmo de posoptimización propuesto y una adaptación del recocido simulado para mejorar los resultados.

Los resultados de la encuesta realizada a los *testers* del grupo Ideti sobre la versión final del complemento muestran un nivel de satisfacción alto con el complemento desarrollado.

Los estudiantes que contaban con experiencia en el desarrollo de pruebas unitarias antes de realizar la prueba alfa, aseguraron volver a utilizar el complemento en comparación con otras herramientas disponibles para soportar la misma tarea y destacaron la velocidad en el tiempo de respuesta y su facilidad de uso una vez hecho todo el proceso con el segundo método probado, dado que entendieron mejor la mecánica de su funcionamiento.

La estrategia de usar el algoritmo de posoptimización, el generar inmediatamente la respuesta y poner en *background* el algoritmo de recocido simulado dio buenos resultados, puesto que permite responder rápidamente, pero además permite encontrar un mejor CA o MCA para futuras pruebas que tengan necesidades similares.

El algoritmo de posoptimización propuesto utiliza solamente 'operaciones válidas', tales que cualquier modificación que se realiza no daña las propiedades de ser CA o MCA, de modo que mientras haya en el repositorio un arreglo capaz de cubrir la necesidad, siempre devolverá un resultado válido y con menos filas. Además, contar con un repositorio con CA y MCA prefabricados, permitió dar tiempo de respuesta aceptables para los *testers.*

Como trabajo futuro se espera entre otras cosas: i) incluir en el complemento la opción de configurar las pruebas con base en datos almacenados en un archivo de Excel, ii) mejorar la arquitectura del complemento y hacerlo disponible al público en general, iii) Hacer nuevas versiones (mejoradas) de los algoritmos de recocido simulado y posoptimización.

AGRADECIMIENTOS

El trabajo en este artículo fue soportado por la Vicerrectoría de Investigaciones de la Universidad del Cauca con el proyecto de investigación VRI-4192.

REFERENCIAS

[1] I. Burnstein, *Practical software testing: a process-oriented approach*, Luxemburgo: Springer Science & Business Media, 2006.

[2] J. Tuya*, et al.*, *Técnicas cuantitativas para la gestión en la ingeniería del software*, As Somozas: Netbiblo, 2007.

[3] A. Mili y F. Tchier, *Software Testing: Concepts and Operations*, Nueva Jersey:Wiley Publishing, 2015.

[4] S. Nidhra y J. Dondeti, "Black Box and White Box Testing Techniques –A Literature Review," *International Journal of Embedded Systems and Applications (IJESA),* vol. 2, pp. 29-50, 2012.

[5] S. Anand*, et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software,* vol. 86, pp. 1978-2001, 2013.

[6] C. P. Jayaswal y T. U. o. T. a. Arlington, *Automated Software Testing Using Covering Arrays*, Texas: University of Texas at Arlington, 2006.

[7] B. S. Ahmed*, et al.*, "The development of a particle swarm based optimization strategy for pairwise testing," *Journal of Artificial Intelligence,* vol. 4, pp. 156-165, 2011.

[8] A. Arcuri y X. Yao, "Search based software testing of object-oriented containers," *Information Sciences,* vol. 178, pp. 3075-3095, 2008.

[9] G. J. B. T. S. C. Myers, *The art of software testing, third edition*. Hoboken, Nueva Jersey: John Wiley & Sons, 2012.

[10] N. Changhai y H. Leung, "A Survey of Combinatorial Testing," *ACM Computing Surveys,* vol. 43, pp. 11-29, 2011.

[11] S. A. Bestoun yK. Z. Zamli, "A review of covering arrays and their application to software testing," *Journal of Computer Science,* vol. 7, pp. 1375-1385, 2011.

[12] H. Avila-George*, et al.*, *Verificación de Covering Arrays*: *aplicando la supercomputación y la computación grid*. Nueva York: Lambert Academic Publishing, 2010.

[13] H. Avila-George y J. Torres-Jiménez, *Construction of Test-Suites*: Omniscriptum Gmbh & Company Kg., 2015.

[14] F. J. Pino*, et al.*, "Using Scrum to guide the execution of software process improvement in small organizations," *Journal of Systems and Software,* vol. 83, pp. 1662-1677, 2010.

[15] D. R. Kuhn*, et al.*, "Software fault interactions and implications for software testing," *Software Engineering, IEEE Transactions on,* vol. 30, pp. 418-421, 2004.

[16] M. Brčić y D. Kalpić, "Combinatorial testing in software projects," presentado en *Proceedings of the 35th International Convention MIPRO*, 2012, pp. 1508-1513.

[17] I. I. Márquez, *Construcción de Torres de Covering Arrays*. [En línea], Disponible: http://www.tamps.cinvestav.mx/defensa_2013_7 , 2013.

[18] M. B. Cohen*, et al.*, "Constructing test suites for interaction testing," presentado en the Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, 2003.

[19] M. B. Cohen*, et al.*, "Constructing strength three covering arrays with augmented annealing," *Discrete Mathematics,* vol. 308, pp. 2709-2722, 2008.

[20] A. Rodríguez-Cristerna and J. Torres-Jiménez, "A Simulated Annealing with Variable Neighborhood Search Approach to Construct Mixed Covering Arrays," *Electronic Notes in Discrete Mathematics,* vol. 39, pp. 249-256, 2012.