COMPLEMENTO DE VS.NET PARA LA DEFINICIÓN DE PRUEBAS DE SOFTWARE DE CAJA NEGRA MEDIANTE ARREGLOS DE COBERTURA

JAIME HERNEY MENESES
EDUAR ALEXIS PEÑA VELASCO

Director: Ph.D. CARLOS ALBERTO COBOS LOZADA
Co-Director: ING. ESP. JIMENA ADRIANA TIMANÁ PEÑA
Asesor: Ph.D. JOSÉ TORRES JIMÉNEZ (CINVESTAV-Tamaulipas, México)

UNIVERSIDAD DEL CAUCA
FACULTAD DE INGENIERÍA ELECTRÓNICA Y TELECOMUNICACIONES
DEPARTAMENTO DE SISTEMAS
GRUPO DE I+D EN TECNOLOGÍAS DE LA INFORMACIÓN (GTI)
ÁREA DE INTERÉS EN SISTEMAS INTELIGENTES E INGENIERÍA DE SOFTWARE
POPAYÁN, MARZO DE 2016

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE ANEXOS

1 INTRODUCCION

1.1 PLANTEAMIENTO DEL PROBLEMA

El software como un sistema compuesto por múltiples componentes puede tener diferentes orígenes de fallos a nivel de comunicación entre sus componentes y/o en su interacción con el usuario. 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. Entre el conjunto de pruebas a realizar están las pruebas de caja negra y las pruebas de caja blanca [1, 2, 3]. Siendo las primeras, las de interés de la presente investigación.

Las pruebas son una de las tareas fundamentales y más costosas en el desarrollo de software. Como se evidencia en [4, 5] el costo de realizar pruebas puede llegar a superar el 50% del costo total del desarrollo del producto, por lo cual la comunidad científica ha enfocado sus esfuerzos en la investigación de técnicas que permitan optimizar este proceso [6]. Aunque los resultados obtenidos han sido gradualmente adoptados por 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 casos de prueba, el costo y el tiempo de su aplicación [4].

Las pruebas buscan evaluar los requisitos funcionales y no funcionales del software apoyándose en casos de prueba de acuerdo con lo especificado en el documento de alcance del proyecto [2]. Una primera aproximación para realizar pruebas consiste en probar todos las posibles interacciones entre las variables o parámetros, conocido como pruebas exhaustivas, lo cual es viable cuando la cantidad de parámetros y sus posibles valores es pequeño, ya que el número de combinaciones puede ser relativamente bajo. Sin embargo, deja de ser viable cuando el número de parámetros y sus valores se incrementa, situación que es más común día a día dada la complejidad creciente de las aplicaciones software, haciendo que las pruebas exhaustivas sean inmanejables debido a incrementos exponenciales en tiempo y costo [3, 7, 8].

Usar un enfoque de pruebas exhaustivas y/o pseudo-exhaustivas [3, 9, 10] requiere evaluar todas las combinaciones de los valores de las variables. Por ejemplo, si se tiene un método con cuatro variables cada una con dos valores posibles (variables binarias), la cantidad de casos de prueba es igual a 16 (2^4), un número pequeño; sin embargo, si el método tiene diez variables binarias, la cantidad es 1024 (2^{10}) combinaciones, lo cual empieza a ser inviable para la prueba de todos los componentes software de un sistema [11, 12]. Una estrategia para manejar esta situación es reducir la cantidad de casos de prueba, de manera que permita reducir la cantidad de tiempo y personal para su ejecución [8, 9, 13].

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 [4]. En [14] 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 [4].

En [4] se muestran numerosas técnicas para realizar pruebas, entre las cuales se mencionan: pruebas al azar, pruebas adaptativas al azar, ejecución simbólica y programa de pruebas de cobertura estructural, generación de casos de prueba basada en modelos, pruebas basadas en la búsqueda y pruebas combinatoriales.

En el marco de las pruebas combinatoriales, se cuenta con objetos matemáticos conocidos como arreglos de cobertura o Covering Arrays (CA) [15] y los arreglos de cobertura mixtos o Mixed Covering Arrays (MCA)[16], que han demostrado ser una estrategia efectiva para la reducción del número de casos de prueba, garantizando un nivel cobertura verificable. Por ejemplo, el NIST (National Institute of Standards and Technology) ha encontrado que usando CA de fuerza 6 (grado de interacción conjunta entre variables) se cubre casi el 100% de los errores en diferentes tipos de aplicaciones software, incluido software de la NASA, software para dispositivos médicos y navegadores de internet[1]. Es así como el uso de CA en las pruebas, permite garantizar la calidad del software con un porcentaje específico de confianza usando el menor tiempo, dinero y esfuerzo posible. Este hecho permite generar y soportar un sello de calidad para las empresas que hacen uso de esta estrategia.

Por otro lado, la realidad de las empresas de desarrollo de software en Iberoamérica, muestra que la gran mayoría de ellas: 1) están conformadas en general por equipos pequeños [17, 18], 2) no cuentan con el personal suficiente para hacer las pruebas de forma exhaustiva, 3) carecen de una herramienta automática para hacer la traducción de los datos de prueba reales a CA y viceversa y que además no está integrada en el Entorno Integrado de Desarrollo (IDE por su siglas en inglés) que usan en la construcción de sus aplicaciones. De allí surge la necesidad de disponer de un complemento de software que se vincule a un IDE y que permita el uso de CA para el diseño de pruebas de caja negra que optimice el proceso de pruebas de calidad en las empresas colombianas.

Por lo anterior, este trabajo de grado buscó una alternativa de solución a una necesidad sentida por las empresas de software: la necesidad de optimizar la tarea de generación de casos de prueba, menos tiempo, menos esfuerzo y la más alta probabilidad posible de detectar fallos que pueda presentar un software. Aunque actualmente existen programas como JUnit o NUnit, para facilitar la pruebas de software, ninguno utiliza los CA y/o MCA como base para crear los casos de prueba, ahí radica la diferencia de este trabajo con el existente, puesto que al usar este tipo de objeto matemático se garantiza que cada dupla, tripleta, cuarteto, quinteto o sexteto de parámetros se pruebe por lo menos una vez, lo que garantiza la detección de hasta el 100% de los fallos según la evidencia empírica mostrada en [10, 19].

En el pasado se ha reconocido la necesidad de crear software de calidad, pero solo hasta ahora se ha hecho un esfuerzo sistemático y continuo para que la industria del software Colombiana genere productos que brinden una real oportunidad de competir en el mercado internacional. En [20] se muestra que en la última década, la industria del software en Colombia ha ganado terreno en el mercado internacional. “A pesar de los esfuerzos, este sector apenas aporta entre el 1.5% y 2% del Producto Interno Bruto, en el

[1] http://csrc.nist.gov/groups/SNS/acts

2009 la industria del Software creció 7.7% frente al 8.9% de Latinoamérica, y aunque estas cifras se pueden mejorar, es necesario que las empresas inviertan en el área de TIC (Tecnologías de la Información y las Comunicaciones)".

Según el diagnóstico del plan Vive Digital [20], Colombia podría desarrollar su sector TI y BPO&O (Business Process Outsourcing & Offshoring o Tercerización de procesos de negocio) comprometiéndose con un programa sectorial de largo plazo que le permita eliminar barreras significativas. La industria de TI globalmente tiene un tamaño de aproximadamente US$ 900,000 millones y crece a un ritmo aproximado de 7% anual. Una exigencia para poder ser competitivo en TI y BPO&O es precisamente cumplir con un estándar mínimo de calidad y desde este trabajo de grado se ofrece un alternativa para lograr ese objetivo.

1.2 JUSTIFICACIÓN

Un CA se caracteriza por tener un número de alfabeto uniforme para todas las columnas (variables), pero en la vida real cada variable puede tener diferente cantidad de valores y por esto se hace necesario usar Arreglos de Cobertura Mixtos (MCA) que permiten diferentes cardinalidades en sus columnas. En los repositorios investigados, solo se tiene disponibilidad de forma libre a los CA, mientras que de los MCA solo se conoce el MCAN (cantidad mínima de filas que posee un MCA) o un reporte del número mínimo de filas encontrados a la fecha, pero no su contenido que es la parte útil para las pruebas de software.

Para contribuir a la solución de las necesidades presentadas anteriormente, en este trabajo se propuso un algoritmo que permite la obtención de un número de casos de prueba manejable para los testers (probadores de software) con el fin de reducir la cantidad de tiempo y personal involucrado en el desarrollo de pruebas software[2] utilizando CA [21] y MCA [11, 22].

Por la naturaleza de las pruebas de software, de tener variables con diferentes cardinalidades (necesidad de MCA), y la no disponibilidad de MCA de forma libre, se propone un algoritmo que permite obtener un MCA (u otro CA de menor cantidad de columnas o menores cardinalidades) de k-columnas, que se adapte a cada prueba, a partir de un CA (o MCA) de k'-columnas, tal que k <= k', y la cardinalidad de cada k_i-columna del MCA (o CA) sea menor o igual a la cardinalidad de su correspondiente columna del CA (o MCA) original. Usando operaciones "válidas" (operaciones que permiten la modificación de la matriz sin dejar de ser un CA o MCA) sobre la matriz C que representa el CA o MCA, como también el uso de un algoritmo voraz para encontrar una solución inicial de forma rápida (buena pero no óptima), y una adaptación del algoritmo de búsqueda meta-heurística conocido como Recocido Simulado o Simulated Annealing (SA)[23] que utiliza más tiempo para su ejecución, pero obtiene mejores resultados.

La elaboración de esta investigación y la implementación del algoritmo propuesto, hace una apropiación de conocimiento altamente especializado hacia la comunidad dedicada al desarrollo de software, en especial a los que tratan el tema de pruebas software. La

[2] En esta investigación, el termino *pruebas de software* hacen referencia a las *pruebas de caja negra*.

utilización de este método podrá reducir significativamente tiempo, dinero y recursos empleados en las pruebas de software permitiendo así contar con software más confiable y seguro. Es preciso comentar que a la fecha no se ha encontrado ningún reporte de investigación con el enfoque planteado en este trabajo en las bases de datos de ACM, IEEE, ScienceDirect, Elsevier y Springer Link. Además, se considera apropiado aplicar el algoritmo propuesto en otras áreas de trabajo como por ejemplo: pruebas de hardware, diseño de experimentos, agricultura, electrónica, medicina, manufactura y en general en cualquier sistema que desee verificar aspectos de calidad y seguridad.

1.3 OBJETIVOS

A continuación se presentan los objetivos tal como fueron aprobados en el anteproyecto por parte del Consejo de Facultad de la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca.

1.3.1 Objetivo general

Proponer un complemento de Visual Studio .NET para soportar la definición de pruebas de software de caja negra mediante el uso de arreglos de cobertura mixtos y la Post-Optimización de arreglos de cobertura homogéneos.

1.3.2 Objetivos específicos

1. Proponer o adaptar un método de Post-Optimización de arreglos de cobertura homogéneos disponibles en el CINVESTAV[3] y en el NIST[4] a arreglos de cobertura mixtos con menor alfabeto o número de parámetros que puedan ser usados en pruebas de software de caja negra.
2. Definir el grado de optimización del algoritmo evaluando el número de filas obtenidas contra el ideal reportado en el CINVESTAV[5], NIST o en el repositorio de Charles Colbourn[6].
3. Elaborar un complemento (plug-in) para Visual Studio .NET, que sea capaz de soportar la generación de casos de prueba automáticos basado en arreglos de cobertura.
4. Definir el nivel de satisfacción de los probadores (testers) de software al usar el complemento propuesto en ParqueSoft Popayán, Visión Software S.A. ó SIIGO - Informática y Gestión S.A. mediante la adaptación[7] de la encuesta presente en encuestafacil.com[8] acerca de la satisfacción del cliente referente a un producto.

1.4 RESULTADOS OBTENIDOS

A continuación se presentan los resultados obtenidos con el desarrollo del trabajo de grado:

---

[3] http://www.tamps.cinvestav.mx/~jtj

[4] http://math.nist.gov/overingarrays/ipof/ipof-results.html

[5] http://www.tamps.cinvestav.mx/~jtj

[6] http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

[7] Se adaptará para medir entre otros los siguientes indicadores: cantidad de pruebas, cantidad de errores detectados, facilidad de uso, tiempo empleado para generar los casos de prueba y el tiempo para introducir los parámetros en la herramienta.

[8] http://www.encuestafacil.com/mas_informacion/plantillas_encuestas_disponibles.aspx

- Repositorio con CAs y MCAs de un máximo de 6 columnas con máximo 6 valores por columna y fuerza 2 y 3.
- Servicio web que permita la conexión entre el complemento desarrollado y el repositorio.
- Complemento software para Visual Studio en sus versiones 2010, 2012 y 2013.
- Monografía donde se presenta el proceso que se llevó a cabo para cumplir con los objetivos planteados en el proyecto y se describen los principales resultados del mismo.
- Un artículo que resume el desarrollo de la investigación y sus resultados.

2 CONTEXTO TEÓRICO

A continuación se describen conceptos fundamentales relacionados con pruebas de software, arreglos de cobertura y arreglos de cobertura mixtos, con el objetivo de facilitar la lectura del resto del documento.

2.1 Pruebas de software

Las pruebas de software[9] (en inglés software testing) se refieren a las investigaciones experimentales y técnicas que tienen como objetivo principal el de proporcionar información objetiva e independiente sobre la calidad del producto software a la parte interesada.

Es una actividad vital para el proceso de control de calidad, ya que es la que busca determinar la confiabilidad del software, siendo así mismo, una de las más complejas y costosas llevadas a cabo durante su desarrollo y mantenimiento. Por lo general estas pruebas se realizan bajo limitaciones de tiempo y presupuesto, lo que fortalece la importancia de diseñar y realizar casos de prueba eficientes que permitan la detección de un porcentaje alto de errores y contribuyan con un proceso de pruebas efectivo (menos costos).

Definición 1 (confiabilidad del software)

> *"Es el funcionamiento sin fallos del software, sobre un intervalo de tiempo dado y bajo ciertas condiciones específicas (requerimientos)"* [1].

Definición 2 (Falla)

> *"Incapacidad de un sistema o componente de software para realizar las funciones requeridas, basado en los requisitos de funcionamiento específicos"* [1].

Definición 3 (Prueba)

> *"Proceso que se utiliza para revelar defectos en el software y para establecer que éste ha alcanzado un grado especifico de calidad con respecto a los atributos seleccionados"* [1].

Definición 4 (Depuración)

> *"Proceso donde se localiza el defecto que es la causa de una falla, se determina la forma de corregirlo, se evalúa el efecto de la corrección y se lleva a cabo dicho cambio"* [1].

Dependiendo del tipo de pruebas de software, estas pueden ser efectuadas en cualquier momento del proceso de desarrollo. Existen diferentes modelos de desarrollo software, así como modelos de pruebas que permiten a cada una pertenecer a diferentes niveles de involucramiento en las actividades de desarrollo.

[9] http://in2test.lsi.uniovi.es/gt26/

La existencia variada de modelos permite el uso de múltiples tipos de pruebas que conllevan a una mejora continua en la calidad del software. Aunque cabe resaltar que por excelentes que sean las pruebas no garantizan las mejores prácticas, lo que indica que no todas son ideales para llevarse a cabo. Por lo tanto, se debe conocer muy bien el contexto sobre el cual se van a trabajar y no caer en el error de aplicar enfoques de pruebas estáticas en pruebas dinámicas o viceversa.

2.1.1 Pruebas estáticas

Proporcionan ayuda sobre la realización de pruebas que no involucran la ejecución de código. Generalmente las pruebas hacen referencia a la revisión de documentos que permiten el hallazgo de fallas en el producto, en el punto más temprano posible del ciclo de desarrollo [24, 25].

2.1.2 Pruebas dinámicas

Constituyen la parte de las pruebas que son realizadas sobre la ejecución de un aplicativo software. Dando uso a técnicas como caja negra y caja blanca que facilitan la medición con mayor exactitud sobre el comportamiento de la aplicación en desarrollo [24, 25]. Dentro de las pruebas dinámicas se pueden mencionar algunas conocidas como las *pruebas de compatibilidad, pruebas de regresión, pruebas de unidad y pruebas de integridad.*

2.1.2.1 Pruebas de compatibilidad

Permiten la comprobación del funcionamiento del software desarrollado en diferentes plataformas como lo pueden ser: sistemas operativos, navegadores, redes y hardware. La mayoría de los programas que se desarrollan no son totalmente nuevos, son a menudo reemplazos de un sistema existente. Algunas veces, estos programas tienen objetivos específicos referentes a su compatibilidad y procedimientos de conversión del sistema existente. Se diseñan los casos de prueba para demostrar que los objetivos de la compatibilidad no se han resuelto y que los procedimientos de conversión no funcionan.

2.1.2.2 Pruebas de regresión

Este tipo de pruebas ratifica el correcto funcionamiento del software desarrollado de acuerdo con evoluciones o cambios funcionales. El objetivo principal consiste en asegurar que los casos de prueba que han sido probados y fueron exitosos, no cambien sus resultados con las modificaciones que se han hecho al software. Para el desarrollo de este tipo de pruebas es aconsejable realizar pruebas automatizadas que garanticen la reducción de tiempo y esfuerzo en su ejecución[26].

2.1.2.3 Pruebas de unidad

Las pruebas unitarias o por unidad son realizadas con la intención de comprobar el correcto funcionamiento de un módulo de código. Con el propósito de asegurar que cada uno de los módulos funcione correctamente por separado. Para posteriormente con las pruebas de integración garantizar el funcionamiento correcto del sistema como un todo [27, 28].

2.1.2.4 Pruebas de integración

Las pruebas de integración son el proceso con el cual los componentes agregados para crear componentes más grandes se evalúan. Es la parte experimental realizada para

mostrar que aunque los componentes hayan pasado satisfactoriamente las pruebas unitarias, la combinación de componentes es incorrecta o insatisfactoria [29].

Para comprender mejor este tipo de pruebas es necesario considerar que las pruebas de integración se pueden realizar a través de una estrategia incremental o big bang.

En la estrategia incremental, cada componente de software primero es probado por separado mediante pruebas unitarias. Posteriormente, se realizan las pruebas de integración con las combinaciones entre diferentes componentes del producto. Formando pequeños grupos de componentes integrados de software que a su vez son probados con otro grupo de componentes ya integrados hasta lograr que todo el sistema haya sido probado como un todo.

Por otra parte, en las pruebas big bang solo se realizan las pruebas de integración, una vez todos los módulos o componentes hayan sido integrados.

2.1.3 Pruebas funcionales

En este caso se realiza la ejecución del aplicativo para evaluar características del componente software. En estas pruebas se busca si la solución satisface las necesidades por la que fue creada, compatibilidad entre versiones, realización del funcionamiento esperado para un grupo de personas, etc. Según las pruebas (más o menos ligeras), se podría hablar de pruebas de regresión, de compatibilidad, de uso a primer nivel (pruebas alfa), pruebas de uso en pre-producción "pruebas beta". Para este caso de estudio se realizaran pruebas alfa y beta.

2.1.3.1 Pruebas alfa

Son realizadas por el cliente en el lugar de desarrollo. El software es utilizado de forma natural con el desarrollador como observador del usuario. Esto se realiza con la intención de obtener un entorno controlado que permita la simulación y/o creación de un ambiente con las mismas condiciones con las cuales se encontraran en las instalaciones del cliente. Una vez logrado esto, se procede a realizar las pruebas y a documentar los resultados [32].

2.1.3.2 Pruebas beta

Son aquellas pruebas realizadas por los usuarios finales. Las pruebas beta vienen después de las pruebas alfa, y se desarrollan en el entorno del cliente, es decir, un entorno que está fuera de control de los desarrolladores. Aquí el cliente se queda a solas con el sistema y trata de encontrarle fallos al producto de los que informa por escrito al desarrollador [30].

2.1.4 Estrategias para la realización de pruebas de software

El personal y/o departamento encargado de las pruebas de software son los responsables de diseñar e implementar los casos de pruebas. Casos de prueba que permitan observar la presencia de fallas y que puedan ser utilizados para la evaluación del rendimiento del software, todo con la intención de utilizar de manera eficiente los recursos disponibles, mediante el desarrollo de un conjunto de casos de prueba que maximicen el rendimiento en la localización de fallas y garanticen una mejora en cuanto al tiempo y el esfuerzo invertido.

Como referencia para el diseño de casos de prueba se tienen las *pruebas de caja negra* y las *pruebas de caja blanca*.

2.1.4.1 Pruebas de caja blanca

También conocidas como pruebas estructurales. Realizan un seguimiento del código fuente según se van ejecutando los casos de prueba, dando paso a la determinación concreta de las instrucciones, bloques, etc. en los que existen errores [1].

Las pruebas de caja blanca cumplen la función de realizar un examen minucioso de los detalles procedimentales, comprobando los caminos lógicos del programa, como también los bucles, condiciones y examinado el estado del programa en varios puntos.

A primera vista, las pruebas de caja blanca realizadas de manera profunda permitirían tener "programas 100% correctos", evaluando todos los caminos lógicos. Desarrollar casos de prueba para todos los caminos lógicos y evaluar los resultados obtenidos supone un estudio demasiado exhaustivo, que prolongaría excesivamente los planes de desarrollo del software “algo poco viable”. Por tal motivo estas pruebas se consideran tediosas, aunque para solucionar este inconveniente se recurre a la implementación de pruebas automatizadas.

“Se dice que las pruebas estructurales o de caja blanca prueban lo que el programa hace y no lo que se supone que debe hacer” [31].

2.1.4.2 Pruebas de caja negra

También conocidas como pruebas funcionales o basadas en la especificación de requisitos y documentación funcional. En estas pruebas se proporcionan las entradas para el software bajo prueba, se realiza la ejecución y se determinan si las salidas producidas son equivalentes a las esperadas [1]. Dicho de otra manera, de estas pruebas interesa la forma de interactuar en el contexto en el que operan (en ocasiones, otros elementos que también podrían ser cajas negras) entendiendo *qué es lo que hace*, pero sin dar importancia a *cómo lo hace*. Por tanto, en las pruebas de caja negra deben estar muy bien definidas sus entradas y salidas, es decir, su interfaz. Por lo que no se precisa definir ni conocer los detalles internos del funcionamiento del componente software que se está probando (**Ver Figura 1**).

Figura 1: *Esquema representativo de caja negra.*

“Se dice que las pruebas funcionales prueban lo que se supone que debe hacer el programa y no lo que el programa hace” [31].

2.1.4.3 Casos de prueba

Las pruebas de software se efectúan mediante casos de prueba. Un caso de prueba detalla los valores de los parámetros de entrada y valores de los parámetros de salida. En este contexto, se asume que un sistema software puede estar conformado por

componentes o módulos, que opera sobre un conjunto de *k* parámetros. Donde cada parámetro recibe un único valor de entrada de un conjunto posible de *v* valores. Cuando cada parámetro tiene distintas cardinalidades, la cardinalidad asociada a cada uno de ellos se indica mediante el vector $v_0\ v_1,\ v_2,\ldots,\ v_{k-1}$.

En un caso de prueba, una configuración indica los valores de entrada que deben ser fijados en cada uno de los *k* parámetros al momento de efectuar una ejecución, un conjunto de pruebas es una colección de casos de prueba. Considerando que los valores posibles para cada parámetro se indican por el vector *v*, la cardinalidad del conjunto de pruebas S que contiene todas las configuraciones posibles de entrada para un componente de software es determinado por la **Ecuación 1**.

$$|S| = \prod_{j=0}^{k-1} v_j \quad \textbf{(1)}$$

Por ejemplo; se desea probar los posibles casos de entrada necesarios para realizar la evaluación de una transacción de retiro en un sistema bancario. A continuación (**ver Tabla 1**) se muestra la cantidad de variables involucradas para el retiro bancario.

Pregunta en cuestión	Opciones de respuesta	Variables involucradas
¿En qué tipo de cuenta se hará el cargo?	Ahorros, Corriente, A Plazos	3
¿La cuenta tendrá saldo?	Saldo = 0, Saldo > 0, Saldo < 0	3
¿Es una cuenta en Moneda Nacional (MN) o Moneda e xtranjera (ME)?	MN, ME	2
¿Pertenece a una Persona natural (PN) o Persona Jurídica (PJ)?	PN , PJ	2
¿La cuenta posee dos o más deudores (mancomunada)?	Si ,No	2
¿En qué estado se encuentra la cuenta?	Activa, Inactiva, Cerrada	3
¿La cuenta permite sobregiros?	Si, No	2
¿De qué tipo será el cargo a la cuenta?	Transferencia entre cuenta propia, transferencia a un tercero, transferencia interbancaria, retiro en efectivo, pago de un cheque	5
¿En qué canal de atención se realizará esta operación?	En ventanilla, cajero Automático – ATM, POS – Pago de servicio o consumo, home banking.	4
Total de configuraciones de entrada		**8640**

Tabla 1: *Total de configuraciones de entrada necesarios para el caso de prueba en el ejemplo del retiro bancario.*

Para este sencillo ejemplo se tendrá |S|=3 x 3 x 2 x 2 x 2 x 3 x 2 x 5 x 4 = 8640 casos de entrada y ni siquiera se está enfocado en los casos de entrada que presentará cada

pantalla. Como menús, listas, grillas, botones etc. Por este motivo se debe delimitar claramente cuál es la funcionalidad que se quiere probar diseñando un conjunto de casos de prueba que permita generar la mayor cantidad de esfuerzo posible al sistema.

Un programa es considerado correcto si se comporta como se espera en todos los posibles casos de entrada, es decir de modo exhaustivo. Usualmente, la cardinalidad de este conjunto suele exceder los recursos disponibles, esto se debe a que el número total de configuraciones crece de manera exponencial.

Definición 5 (Dominio de entrada)

> *"El conjunto de todas las posibles entradas (configuraciones) a un programa es denominado como dominio de entrada o espacio de entrada"* [1].

Para comprender mejor el incremento exponencial en el dominio de entrada, se tomará como referencia el componente software "**Advanced Task Killer**" mostrado en la **Figura 2.** Puede apreciarse que todas las opciones que tienen los valores poseen dos estados: seleccionado (cuadro con un check) y no seleccionado (cuadro en blanco). Es decir que la cardinalidad de los parámetros es de 2 (*v=2)*, La **Ecuacion 1** mostrada anteriormente, se puede enunciar de la siguiente manera $|S|= v^k$. Para este ejemplo existen 2^{10} = 1024 configuraciones de entrada, sin embargo, si se agregaran 4 parámetros adicionales (*k=14*), esta cantidad incrementaría llegando a 2^{14} = 16,384.

2.2 Pruebas exhaustivas

Pruebas exhaustivas son pruebas que abarcan todas las combinaciones de valores de entrada y precondiciones. Es decir exponer un sistema a todas las situaciones posibles, para garantizar que se encontró hasta el último fallo. Permitiendo así que se garantice una respuesta a cualquier caso que se presente en la ejecución real del mismo [29].

2.3 Pruebas pseudo-exhaustivas

Las pruebas pseudo-exhaustivas utilizan la observación empírica como principal elemento para la detección de fallos en un componente software. Estas pruebas logran cubrir la cantidad total de pruebas sin necesidad de recurrir a la comprobación de todas las combinaciones de valores de entrada y precondiciones. Es decir, la observación empírica sugiere que el número de variables que intervienen en los fallos de software es relativamente pequeño, (del orden de 3 a 6), al menos para algunos tipos de software. Por lo tanto, si se sabe por experiencia que t o menos variables están involucradas en los fallos para un tipo de aplicación particular y se puede probar todo con (t + 1) combinaciones de variables, es posible garantizar que la aplicación funcionará correctamente. Visto de otro modo, si se sabe de antemano que todos los fallos son provocados por t o menos condiciones, probando todas las condiciones t de entrada en cierto sentido es equivalente a una prueba exhaustiva. Esta es la forma como las pruebas pseudo-exhaustivas se valen de ciertas estrategias para cumplir con el objetivo de encontrar la mayor cantidad de errores posibles y garantizar el buen funcionamiento de un objeto en prueba. Aquí es donde los arreglos de cobertura (covering arrays, CA que se explican en una sección posterior) toman gran valor.

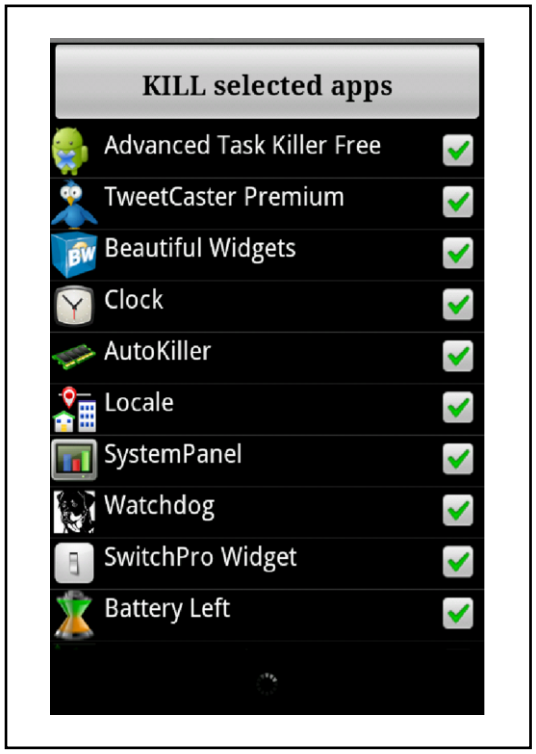


Figura 2: *Aplicación Advanced Task Killer (*ATK*), usada para matar las aplicaciones en ejecución del sistema operativo Android, tomada de [1].*

2.3.1 Pruebas combinatoriales

Las pruebas combinatoriales asumen el software como una caja negra con entradas y salidas que deben corresponder. Las diferentes variables de entradas reciben valores desde el exterior del programa, los cuales se utilizan con diversas combinaciones para generar salidas específicas. Para realizar las pruebas, se considera el software como un clasificador (**ver Figura 3**) que permite separar todas las posibles combinaciones de entrada en unas pocas categorías, donde cada una de estas se ejecuta como un subprograma independiente de los otros. Dicho de otra manera es como si se tuviera un conjunto reducido de funciones las cuales se eligen al entrar y una vez iniciado, ya no existen caminos alternos, por lo tanto se debe seguir hasta terminar. Para ejemplificar un poco en un contexto de programación, es como utilizar una estructura condicional múltiple para determinar un flujo de salida (opción) para cada entrada en cuestión [32].

2.3.1.1 Arreglos de Cobertura (CA)

Sean N, k, v, t enteros positivos. Un arreglo de cobertura, CA(N;k,v,t), de alfabeto v, fuerza t, es un arreglo de tamaño N × k, donde cada elemento $a_{i,j}$ toma como valor un símbolo del conjunto S = 1, 2, ..., v, tal que cada sub-arreglo N × t contiene todas las posibles combinaciones de los v^t símbolos al menos una vez. En la actualidad muchos covering arrays aún son desconocidos; sin embargo, en la literatura se reportan el mínimo teórico para un determinado CA llamado *Lower Bound* y el mínimo para el cual existe un arreglo conocido y se define como *Upper Bound*. Cuando existe un arreglo cuyo N es el lower bound, a ese tamaño se conoce como CAN (**Covering Array Number**) [11, 33-35].

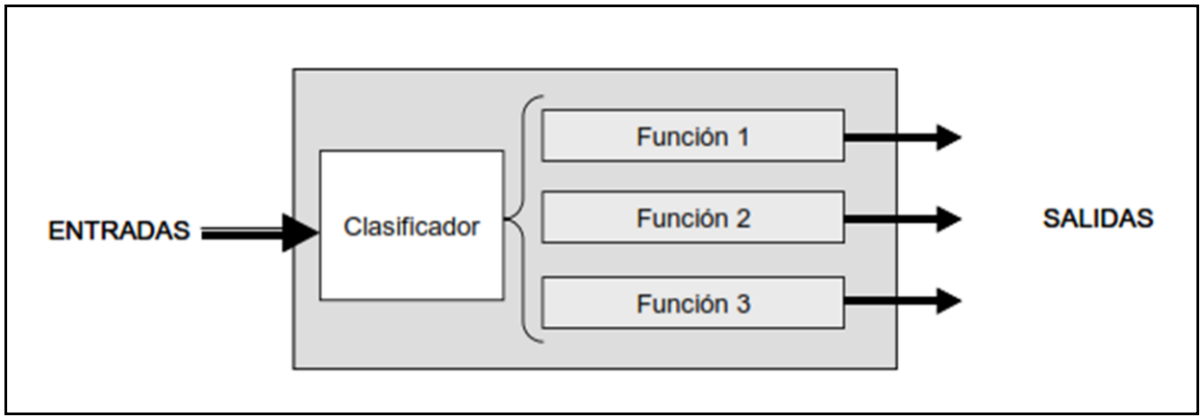


Figura 3: *Pruebas combinatorias, tomado de [32].*

Dadas las características de los CA (existencia, máxima cobertura y mínima cardinalidad), éstos han sido rápidamente adoptados por la comunidad relacionada con las pruebas de software. Para ilustrar como se puede usar un CA (o MCA) para la realización de una prueba de software, supongamos que t=2 (nivel de interacción entre los parámetros), 5 factores/parámetros (uno con tres valores y cuatro con dos valores). Los posibles valores del primer factor de la aplicación software que se va a probar se relacionan con el manejador de base de datos que puede usar (MySQL, Sybase y Oracle). Los posibles valores del segundo factor se relacionan con el sistema operativo en el que se ejecuta el producto (Windows y Linux). Los posibles valores del tercer factor se relacionan con el navegador utilizado para acceder a la aplicación (Internet Explorer y Firefox). Los valores del cuarto factor se relacionan con el protocolo usado para comunicarse con el servidor de la aplicación (IPv4 o IPv6). Los valores del quinto factor se relacionan con el procesador del servidor (Intel o AMD). La **Tabla 2** muestra los experimentos (casos de prueba) requeridos por la aplicación usando un CA óptimo, el cual requiere sólo 6 experimentos. Si se hicieran pruebas exhaustivas se requerirían 48 experimentos (3*2*2*2*2).

Experimento / Factor	Base de Datos	Sistema Operativo	Navegador	Protocolo	Procesador
Caso de Prueba 1	MySQL	Windows	Internet Explorer	IPv4	Intel
Caso de Prueba 2	MySQL	Linux	Firefox	IPv6	AMD
Caso de Prueba 3	Sybase	Windows	Firefox	IPv6	Intel
Caso de Prueba 4	Sybase	Linux	Internet Explorer	IPv4	AMD
Caso de Prueba 5	Oracle	Windows	Internet Explorer	IPv6	AMD
Caso de Prueba 6	Oracle	Linux	Firefox	IPv4	Intel

Tabla 2. *CA para realizar pruebas de software de una aplicación con 5 factores, fuerza 2, tres valores en el primer factor/parámetro y dos valores en el resto de factores, tomado de* [36].

2.3.1.2 Arreglos de Cobertura Mixtos (MCA)

Un Mixed Covering Array (MCA) es una generalización de un CA que permite diferentes alfabetos en diferentes columnas. Un MCA elimina una limitación que se evidencia en los CA, relacionada con el hecho de que “todos los parámetros deben tener el mismo número de valores”. Pero como se mostró en el ejemplo anterior, en el contexto de pruebas software lo más común es que los valores para cada parámetro de entrada tengan diferente cardinalidad, por esto los MCA son las herramientas más usadas en este contexto.

Un Mixed covering Array, denotado como MCA(N; t, k, (v_0, v_1,…, $v_{(k-1)}$)). Es una matriz de dimensión N x k, donde v_0, v_1,…, $v_{(k-1)}$ es un vector de cardinalidad que indica los valores para cada columna (v= $\sum_{j=0}^{k-1} v_j$). Un MCA tiene las siguientes propiedades:

1. Cada columna j (0≤j≤k-1) contiene únicamente símbolos del conjunto Sj, donde |Sj|=v_j.
2. Las filas de cada sub-arreglo N × t cubren todas las t-tuplas de valores de las t columnas al menos una vez. El mínimo N para la cual existe un MCA se llama Mixed Covering Array Number MCAN(t, k, (v_0, v_1,…, $v_{(k-1)}$)). Una notación manejable para un MCA puede ser usando la notación exponencial MCAN(t, k, ((v_0)^{q0}, (v_1)^{q1},…, (v_w)^{qw})); esta notación describe que hay q_r parámetros del conjunto {v_0, v_1,…, $v_{(k-1)}$} que toma valores de v [16].

Para ilustrar un enfoque de una MCA aplicado en el diseño de pruebas software, se considera el ejemplo del sistema de comercio electrónico mostrado en la **Tabla 3**. El ejemplo implica cuatro parámetros, los dos primeros parámetros tienen 3 posibles valores y los últimos dos parámetros tienen sólo 2 valores posibles; para probar el software de manera exhaustiva es necesario un conjunto de 3 × 3 × 2 × 2 = 36 casos de prueba.

Browser	Web Server	Database	Payment
Firefox	Apache	MySQL	Visa
Chromium	IIS	MaxDB	MasterCard
IE	WebSphere		

Tabla 3. *Pruebas combinatoriales usando Mixed Covering Arrays, un ejemplo de sistema basado en la Web (parámetros) [16].*

El uso de un MCA (con interacción de tamaño t = 2) requerirá sólo 9 casos. La **Figura 4** (a) muestra un Mixed Covering Array correspondiente a MCA (9; 2, 4, 3^2 2^2). Por último, para hacer el mapeo de la Mixed Covering Arrays y el sistema de comercio electrónico, cada posible valor de cada parámetro en la **Tabla 3** se marca por el número de fila. La **Figura 4 (b)** muestra el conjunto de pruebas por parejas correspondiente con cada uno de sus nueve experimentos. Es similar a una fila del Mixed Covering Arrays mostrada en la **Figura 4 (a)**.

(a)

0	0	0	0
2	1	0	1
1	2	0	1
0	2	1	0
2	0	1	1
1	1	1	0
0	1	0	1
2	2	1	0
1	0	1	1

(b)

1	Firefox	Apache	MySQL	Visa
2	IE	IIS	MySQL	MasterCard
3	Chromium	WebSphere	MySQL	MasterCard
4	Firefox	WebSphere	MaxDB	Visa
5	IE	Apache	MaxDB	MasterCard
6	Chromium	IIS	MaxDB	Visa
7	Firefox	IIS	MySQL	MasterCard
8	IE	WebSphere	MaxDB	Visa
9	Chromium	Apache	MaxDB	MasterCard

Figura 4: *Ejemplo de un sistema basado en la Web utilizando pruebas combinatoriales con Mixed Covering Arrays. En (a) Se muestra un MCA(9; 2, 4, 3^2 2^2) para el sistema de comercio electrónico. En (b) se muestra el repositorio de pruebas que cubre todas las interacciones para MCA (9, 2, 4, 3^2 2^2), tomada de [16].*

2.4 Estudio experimental del NIST

Un grupo de investigadores del Instituto Nacional de Estándares y Tecnología de EE.UU. (NIST) estudiaron durante 15 años el costo generado por la presencia de fallos en sistemas software, para este estudio se involucró un sistema embebido para dispositivos médicos, un navegador web, un servidor web (apache) y un sistema de base de datos. Como se evidencia en [5, 37], los investigadores del NIST contaron el número de factores individuales que habían intervenido en las fallas subyacentes y los fracasos reales; en los cuatro sistemas investigados encontraron que: 29-68% de las fallas habían involucrado un solo factor (parámetro); 70-97% involucraron uno o dos factores; 89-99% involucraron tres o menos factores; 96-100% involucraron cuatro o menos factores; 96-100% involucraron cinco o menos factores, y ningún fallo había involucrado más de seis factores (**ver Figura 5**).

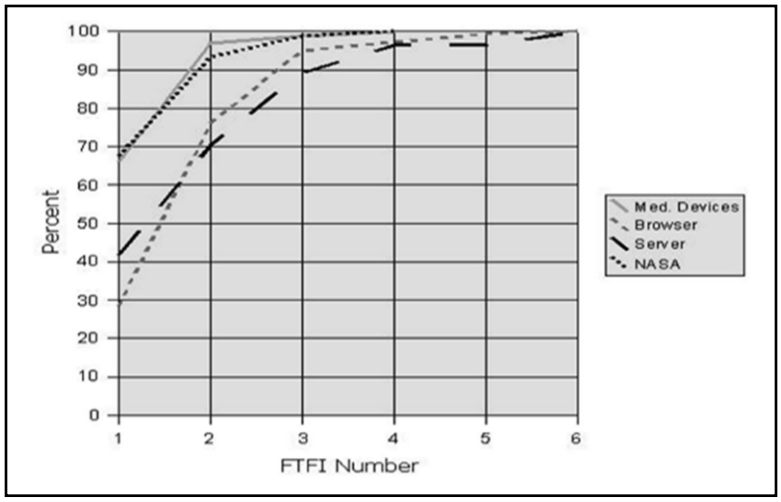


Figura 5: *Porcentaje de fallas cubiertas en 4 sistemas distintos con distintas combinaciones de factores/parámetros (Failure-triggering fault interaction, FTFI), tomado de [10, 19].*

Los datos mostrados en la **Figura 5** sugieren una regla de interacción como la siguiente: sólo algunos factores están involucrados en fallos del software. La mayoría de las fallas son inducidas por un factor único o por el efecto combinado (interacción) de dos factores [33] y casi el 100% de las fallas se encuentran revisando el efecto combinado de un máximo de seis factores.

3 ESTADO DEL ARTE

El software es un conjunto de múltiples componentes que interactúan entre sí para lograr un objetivo. Un fallo en alguno de estos componentes puede generar fallos en toda la aplicación, por tanto se hace necesario realizar pruebas que permitan detectar errores y corregirlos para asegurar la calidad del producto. Probar solo algunos valores en un componente software puede no ser suficiente, pero probar todas las combinaciones posibles (pruebas exhaustivas) puede llegar a ser demasiado costoso e inmanejable. Un balance entre los dos enfoques consiste en generar un conjunto de casos de prueba de tamaño razonable que permita detectar la gran mayoría (o todos) los errores (pruebas pseudo-exhaustivas).

Una primera aproximación para realizar pruebas de software pseudo-exhaustivas lo constituye la estrategia de pruebas basadas en el azar [38] que consiste en tomar aleatoriamente unas combinaciones (pseudo-aleatorias) para realizar las pruebas. Esta estrategia puede dar buenos resultados en algunos casos donde un probador humano no tenga acceso a otro método, es a menudo considerada como la peor estrategia, ya que no asegura cubrir la mayoría posible de interacciones, pudiendo además repetir muchos casos de prueba y dejar otros por fuera debido a su naturaleza aleatoria [38]. Otro problema es que no utiliza la información disponible para orientar el desarrollo de casos de prueba.

En [39] White and Cohen, analizan ciertos tipos de fallo de programas numéricos y observaron que cuando el contenido de predicados (puntos de toma de decisiones en el código fuente) son erróneos, se toma un cálculo de ruta incorrecta (referido como errores de dominio). Esto a menudo resulta en “regiones contiguas” del dominio de entrada que revelan fracasos. Ellos propusieron una técnica sistemática para detectar este tipo de errores, conocido como pruebas adaptativas basadas en el azar, las cuales toman ventaja del fenómeno mencionado y crean los casos de prueba de tal forma que se distribuye su cantidad de una forma más uniforme en todo el dominio de entrada.

En [40] se propone el uso del algoritmo de búsqueda tabú para automatizar el proceso de generación de casos de prueba en un enfoque de caja negra, el cual permite trabajar sobre las especificaciones del programa y su implementación, como también en las pruebas de caja blanca, las cuales son ejecutadas desde el programa.

En [41] se muestra una definición y la aplicación de pruebas simbólicas, junto con sus ventajas y desventajas. Además en [42] se muestra un algoritmo basado en modelos que genera un conjunto de pruebas que cubre todos los elementos factibles sin incluir caminos repetidos. Y en [14] se muestran diferentes algoritmos de búsqueda para la construcción de casos de prueba.

Otro enfoque más reciente conocido como Pruebas Combinatoriales (Combinatorial Testing), busca la construcción de pequeños casos de prueba funcionales, que proporcionan la cobertura de las configuraciones más comunes [6] [19, 43, 44]. En [33] se muestran los objetos combinatorios más usados en el diseño de experimentos. Entre ellos se destacan los Cuadrados Latinos (Latin Squares, LA), los arreglos ortogonales (Orthogonal Arrays, OA) y los arreglos de cobertura (Covering Arrays, CA) con sus respectivas modificaciones. Cada uno de estos objetos han sido usados en diferentes disciplinas. En [45] se muestran aplicaciones de los OA en el área de las matemáticas

discretas, y el uso de CA que han sido aplicados para realizar pruebas funcionales en diversas áreas, entre ellas, las pruebas de componentes de software. El uso de los CA permite probar todas las interacciones, de un determinado tamaño, entre los parámetros de entrada, utilizando un menor número de casos de prueba y garantizando que cada combinación (pares, tripletas, cuartetos, quintetos, sextetos, según la fuerza de interacción con que se creen los CA) de parámetros se prueba por lo menos una vez.

3.1 Arreglos Ortogonales (OA) y Arreglos de Cobertura (CA) [36]

Arreglos Ortogonales: Sean N, k, v, t enteros positivos. Un arreglo ortogonal, $OA_\lambda(N; k, v, t)$, de índice λ, alfabeto v, fuerza t, es un arreglo de tamaño N × k, donde cada elemento $a_{i,j}$ toma como valor un símbolo del conjunto S = 1,2,…,v, tal que cada sub-arreglo N×t contiene todas las posibles combinaciones de los v^t símbolos exactamente λ veces. Un OA tiene la propiedad λ= N/vt, razón por la cual se puede dejar fuera la N de la notación, y expresarlo de la siguiente manera: OA(k, v, t). Cuando se suprime λ de la notación se asume que el valor del índice es 1. Los primeros trabajos realizados, donde se aplicaron los objetos combinatorios al diseño de pruebas, fueron hechos en disciplinas como la medicina, la agricultura y la manufactura [46].

Arreglos Ortogonales y Pruebas de Software: En la **Tabla 4** , se muestra un arreglo ortogonal con 8 renglones (cada renglón es un experimento) involucrando 5 factores, un factor con 4 valores (0, 1, 2, y 3), y 4 factores con dos valores (0 y 1). La interacción es de tamaño dos (t=2). Puede validarse por ejemplo que si se toma el factor con 4 valores (Factor 1) y un factor con dos valores (ej. Factor 2) las combinaciones (0,0), (0,1), (1,0), (1,1), (2,0), (2,1), (3,0) y (3,1) aparecen exactamente una vez, de manera adicional puede validarse que para cualquier selección de dos factores con dos valores (en total hay seis parejas de factores con dos valores) existen dos ocurrencias de las combinaciones (0,0), (0,1), (1,0) y (1,1).

Experimento / Factor	Factor 1	Factor 2	Factor 3	Factor 4	Factor 5
Experimento 1	0	0	0	0	0
Experimento 2	0	1	1	1	1
Experimento 3	1	0	0	1	1
Experimento 4	1	1	1	0	0
Experimento 5	2	0	1	0	1
Experimento 6	2	1	0	1	0
Experimento 7	3	0	1	1	0
Experimento 8	3	1	0	0	1

Tabla 4. *Un arreglo ortogonal con 8 experimentos 5 factores y una interacción de tamaño 2 (t=2), tomado de [36].*

Un problema de los OA es que puede dar lugar a pruebas excesivamente grandes con λ> 1. Para los casos de v y k donde exista un OA con λ= 1, éste sería el conjunto óptimo de pruebas. Sin embargo, hay muchos valores de v y k donde los OA con λ = 1 no existen, por ello se tiene que recurrir a una estructura menos restrictiva conocidas como los arreglos de cobertura (CA).

3.2 Construcción de Arreglos de Cobertura

La construcción de CA es una tarea compleja, en [14, 33, 40] se muestran métodos para realizar esta tarea utilizando diferentes enfoques y algoritmos. En [11] se muestra una revisión de su aplicación y construcción, destacando entre las estrategias de construcción dos grandes grupos que son: 1) un parámetro al tiempo (one-parameter-at-a-time) y 2) una fila al tiempo (one-row-at-a-time).

En el primero (one-parameter-at-a-time) se agrega un parámetro (columna) a la matriz y se comprueba las t-interacciones y luego un algoritmo voraz elige los valores de los parámetros que puedan cubrir más interacciones, entre estos algoritmos se encuentran: IPOG [47] y sus mejoras, IPO-s [48], IPOG-F e IPOG-D [49]. Además en [50] se propone una estrategia basada en árboles con este enfoque.

Considerando el caso de una sola fila al tiempo (one-row-at-a-time). Además de utilizar el algoritmo voraz, la estrategia construye una fila y comprueba su cobertura. Las filas que soportan más interacciones son elegidas para formar la matriz final. Entre los algoritmos que usan este enfoque se encuentran: The Automatic Efficient Test Generator (AETG) [51], MAETG [52] Pairwise Independent Combinatorial Testing (PICT) [53], Deterministic Density Algorithm (DDA) [54], Classification-Tree Editor eXtended Logics (CTE-XL) [55, 56] y Test Vector Generator (TVG) [57].Post-Optimización Recientemente, la construcción de CA se ha visto como un problema de optimización. Como parte del método de construcción de una sola fila al tiempo, se usan meta-heurísticas que buscan el número óptimo de filas. Diferentes métodos de optimización se utilizan para la construcción de CA con esta estrategia. En los resultados publicados, se ha mostrado que estas estrategias pueden lograr menores tamaños en la gran mayoría de los casos, pero con un tiempo de construcción más largo que otras estrategias [58]. Hasta el momento, el recocido simulado (SA) [37, 52], los algoritmo genéticos (GA) [59], el algoritmo basado en colonias de hormigas (ACA) [59], la búsqueda tabú (TS) [35] y la optimización por enjambres de partículas [60] se han aplicado con éxito para construir CA con una fuerza de interacción pequeña.

En la literatura revisada se encontraron dos reportes de técnicas de Post-Optimización [61]. Estas técnicas consisten en la eliminación de parejas repetidas en un CA y al mismo tiempo garantizando una cobertura total, permitiendo una reducción de hasta el 10% de las filas del CA original. El CA obtenido tiene la misma cobertura que el original pero se tiene menor cantidad de casos de prueba. los métodos como recocido simulado [62] y búsqueda tabú [63] han encontrado CA más óptimos pero con mayor gasto de tiempo.

3.3 Los CA en pruebas de software

Herramientas software que usan CA: En [5] se muestra una tabla comparativa de dos herramientas, FireEye y Tconfig, que construyen CA para aplicación en pruebas de software, ambas son programas comerciales desarrolladas en java que no están integrados en ningún IDE (Entorno Integrado de Desarrollo), además realizan la construcción de CA desde cero, por tanto sus tiempos para manejar un problema grande pueden llegar a ser altos. Lo anterior se soluciona con la presente propuesta, donde se define un complemento para Visual Studio .NET (IDE de Microsoft) haciendo uso de un CA que es el resultado de la Post-Optimización de otros CA previamente construidos y disponibles en un repositorio (ahorrando tiempo de construcción).

Repositorios disponibles para el trabajo con CA: En la literatura se mencionan tres repositorios importantes, a saber:

- Repositorio de Charles Colbourn[10], que presenta únicamente las mejores cotas de los CA sin dar explícitamente los casos de prueba.
- Repositorio del NIST[11], que permite descargar CA construidos con el método IPOG.
- Repositorio del CINVESTAV[12], que permite descargar CA construidos usando la meta heurística de recocido simulado. Aquí se presentan CA optimizados del repositorio del NIST y otros nuevos. Con acceso restringido a partir de octubre de 2015.

[10] http://www.public.asu.edu/λccolbou/src/tabby/catable.html
[11] http://math.nist.gov/overingarrays/ipof/ipof-results.html
[12] http://www.tamps.cinvestav.mx/~jtj

4 PROPUESTA

Partiendo del hecho de la inexistencia de un complemento para VS.Net que permita la generación de casos de prueba de caja negra, se toma como referencia la utilización de CA pertenecientes a los repositorios del CINVESTAV, NIST y Colbourn. Donde cada CA utilizado es sometido a un proceso que garantice la reducción de CA homogéneos a CA mixtos (MCA) o CA homogéneos de menor dimensión, según sea requerido. A continuación se muestra el esquema general de la solución propuesta (**ver Figura 6**).

4.1 Prototipo del sistema

El complemento de Visual Studio.Net se desarrolló por componentes como son: el algoritmo de Post-Optimización, la interfaz gráfica de usuario (GUI), el codificador-decodificador, un repositorio que consiste en una base de datos SQL-Server 2008 y un servicio web que permite la comunicación entre el complemento y el repositorio. Para implementarlo se siguió las instrucciones (mejores prácticas) dadas por Microsoft[13].

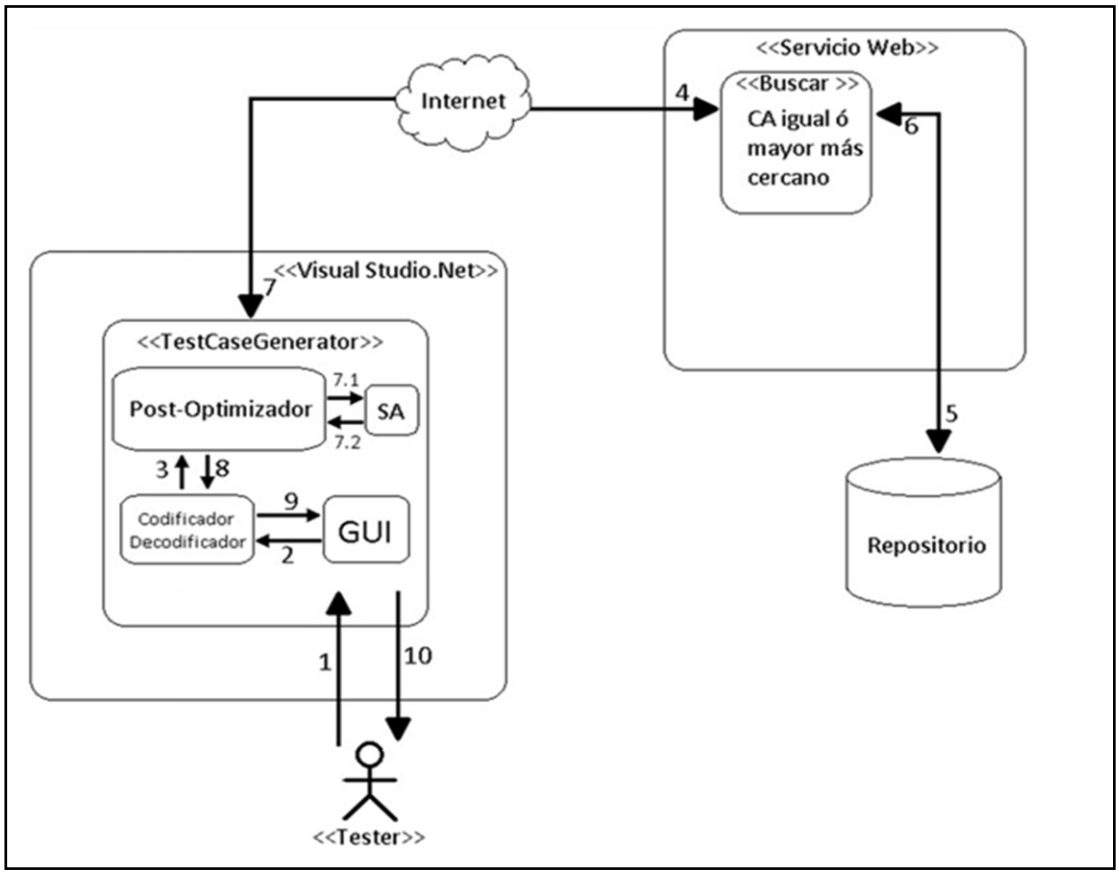


***Figura 6**: Esquema del prototipo.*

La **Figura 6** muestra de forma general una serie de pasos que se siguen para obtener un conjunto de casos de prueba utilizando el complemento, a continuación muestra en que momento sucede cada uno de ellos:

[13] https://support.microsoft.com/es-es/kb/317345

1. Una vez iniciada la sesión por el tester, la interfaz principal (GUI) permite buscar y cargar un archivo .exe o .dll escrito en C# que contiene las clases con los métodos que se van a probar. Posteriormente, el tester selecciona el método sobre el cual desea generar los casos de prueba. Seguidamente la interfaz importa el nombre de la clase, nombre del método, parámetros con sus tipos, el tipo de retorno y si es o no estático, finalmente permite configurar dos o más valores por cada parámetro de entrada y la fuerza de interacción deseada (2 - 6).
2. Teniendo configurada la prueba, se usa la cantidad de parámetros con sus valores y se transforman en un alfabeto deseado utilizando el codificador-decodificador.
3. El codificador-decodificador entrega al post-optimizador el alfabeto deseado.
4. El post-optimizador realiza una búsqueda de un CA o MCA con alfabeto igual al que necesita o uno mayor más cercano utilizando el servicio web para comunicarse con el repositorio.
5. El servicio web realiza la búsqueda en el repositorio.
6. El repositorio devuelve el CA o MCA que encuentre.
7. El post-optimizador recibe un CA o MCA y lo almacena en memoria, en este momento pueden suceder dos casos:1) se encontró uno exacto, entonces se guarda temporalmente para poder usarlo tal como está, y 2) se encontró uno cercano, entonces se aplica el proceso de post-optimización, seguido del recocido simulado (paso 7.1 y 7.2) y queda listo para ser usado.
8. Teniendo el CA o MCA requerido se transforma de nuevo en casos de prueba mediante el uso del codificador-decodificador quien reemplaza cada fila del CA o MCA en un caso de prueba con los valores configurados. Y a partir de los casos de prueba permite al tester visualizar y configurar los valores esperados y las variables de salida si estas existen en el método. Finalmente, después de guardarlas, se genera el código fuente para copiar y pegar directamente sobre un proyecto de pruebas.

A continuación se describe con mayor detalle cada uno de los componentes que conforman el prototipo del sistema.

4.1.1 Interfaz Gráfica de Usuario - GUI

La interfaz de usuario está conformada por un conjunto de objetos gráficos, los cuales cumplen con la función de representar información, mostrar las acciones disponibles y mejorar la interacción con el tester. En esta investigación las interfaces pertenecientes al complemento se muestran cómo: inicio de sesión, configuración de la prueba, configuración de valores esperados y cierre de sesión.

4.1.2 Codificador - Decodificador

Es un mediador que pre-procesa la información recibida mediante la interfaz gráfica de usuario y la transforma en el lenguaje apropiado, es decir, configura el alfabeto y la fuerza de interacción para posteriormente realizar una búsqueda del CA o MCA. Si la búsqueda es correcta, hace uso del encontrado, de lo contrario busca el mayor más cercano, lo adapta al CA o MCA requerido mediante el post-optimizador y lo aplica para entregar un resultado en tiempo real, mientras tanto se ejecuta en de forma asíncrona el recocido simulado (SA). Para disminuir aún más la cantidad de filas y poder usarlo más adelante en otras pruebas que requieran la misma cantidad de columnas y la misma cantidad de valores por cada una de las columnas.

4.1.3 Post-Optimizador

Es el encargado de adaptar el CA o MCA de acuerdo con la inicialización de las variables. Y así proceder con la configuración de los casos de prueba. Más adelante en el **numeral 4.2** se indica con mayor detalle el proceso que se sigue para la Post-Optimización.

4.1.4 Recocido simulado (SA)

Permite mejorar el resultado encontrado por el post-optimizador y obtener un CA o MCA con menor número de filas, pero a cambio de incrementar su costo de ejecución. En el **numeral 4.3.1** se detalla la adaptación del algoritmo que se utilizó.

4.1.5 Servicio Web

Es el encargado de la intercomunicación entre el complemento y el repositorio. Cumple con la funcionalidad del inicio y cierre de sesión, así como también la realización de operaciones DML sobre la base de datos (SQL server 2008) necesarias para garantizar la portabilidad del complemento y avalar su uso desde cualquier parte del mundo.

4.1.6 Repositorio

Es el sitio centralizado donde se almacena y mantiene la información correspondiente con CA y MCA sobre la base de datos. Así como también los usuarios con sus credenciales, la configuración de las pruebas y sus correspondientes casos de prueba.

4.2 Proceso de Post-Optimización

El proceso 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 siguientes pasos:

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 objetivo (V2), dejando en cada columna sólo 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.

    a. Crear o actualizar la lista de combinaciones repetidas (listaRepetidos). Para hacer esto, se usa el algoritmo TestCA (**ver Figura 7**), el cual calcula la matriz P, que almacena la cantidad de veces que una combinación de valores de parámetros en combinaciones de columnas ocurre. Todo valor P_{ij} cuyo valor sea mayor que 1 es redundante y se utiliza para crear la lista de combinaciones repetidas.
    b. Buscar comodines en la fila actual de la matriz C basándose en la lista de combinaciones repetidas (**ver Figura 8**).

c. Borrar las filas que tienen más de K - t comodines, es decir que no cumplen con la fuerza t (**ver Figura 9**).
d. Borrar filas repetidas y ordenar por cantidad de comodines. Ubicando las filas con la mayor cantidad de comodines al final (**ver Figura 10**).
e. Tomar parejas de filas y combinarlas si se puede. Dejando como resultado una fila que representa a las dos en caso de poder combinarse (**ver Figura 11**).

```
TestCA : booleano
listaIndices ← Ø
listaValoresRepetidos ← Ø
J ← construirJ(k,t) // Genera una lista con el polinomio mayor que
P ← Ø
esCA ← verdadero
max ← CombinacionesDeKenT(k,t)
para i ← 0 hasta max con paso 1 hacer
tamañoP ← MayorDimensionP(v, Ji)
      vectorP ← Ø
      para fila ← 0 hasta n con paso 1 hacer
            posicionEnP ← Divisionsintetica(v, C, fila, Ji)
            si posicionEnP ≠ -1 entonces
                  vectorPposicionEnP ← vectorPposicionEnP + 1
            fin si
      fin para
      P ← adicionar (vectorP)
      para pos ← 0 hasta tamaño Con paso 1 hacer
            si vectorPpos ← 0 entonces
                  esCA ← falso
            fin si
            si vectorPpos > 1 entonces
                  x ← MultiplicacionSintetica(v, Ji, pos, t)
                  posición ← Ji
                  valor ← x
                  para pos ← 0 hasta tamañoP Con paso 1 hacer
                        listaIndices ← adicionar(posicion);
                        listaValoresRepetidos ← adicionar(valor);
                  fin para
            fin si
      fin para
fin para
retornar esCA
```

Figura 7: *Algoritmo TestCA, calcula la matriz P, que almacena la cantidad de veces que una combinación se encuentra en la matriz C. El método DivisionSintetica[14] mapea una t-tupla a una columna en P y el método MultiplicacionSintetica realiza el proceso inverso al de la división.*

[14] Se utiliza la división sintética para encontrar un índice del vector P. Para posteriormente determinar el número de repeticiones que aparece una combinación. Y determinar si se está en presencia de un CA o no. Se determina que no se encuentra un CA, cuando hace falta al menos una combinación en el vector P.

```
metodoBuscarComodines: entero
Entradas Matriz C
actual ← final
filaModificada ← -1
contador ← 0
comodinesEncontrados ← 0
mientras actual > = primero y comodinesEncontrados = 0 hacer
      filaActual ← C_actual
      para columna ← 0 hasta k Con paso 1 hacer
            contador ← 0
            listaCombinaciones = combinarColunmna(FilaActual, columna)
            para i ← 0 hasta contar(ListaCombinaciones)Con paso 1 hacer
                  combinacion ← listaCombinaciones_i
                  si estaRepetida(combinacion) entonces
                        contador ← contador+1
                  fin si
            fin para
      fin para
      si contador = contar(ListaCombinaciones)y contador ≠ 0 entonces
            comodinesEncontrados ← comodinesEncontrados +1
            cFilaActual,columna ← comodin
            filaActualcolumna ← comodin
            filaModificada ← actual
      fin si
fin mientras
retornar filaModificada
```

Figura 8: *Algoritmo para la búsqueda de comodines.*

```
metodoBorrarFilas:
Entradas entero n, entero k, entero t
para i ← 0 hasta n Con paso 1 hacer
      contador ← 0
      para j ← 0 hasta k Con paso 1 hacer
            si Cij ← comodin entonces
                  contador ← contador+1
            fin si
      fin para
      si contador > (k – t) entonces
            borrar_Fila()
      fin si
```

Figura 9: *Algoritmo para borrar filas.*

```
metodoBorrarFilasRepetidasYordenarPorCantidadDeComodines:
para i ← 0 hasta k+1 Con paso 1 hacer
      indicesSegunComodin ← Ø
fin para
comodinesEnFila ← 0
para i ← 0 hasta n Con paso 1 hacer
      fila ← obtenerFila(C, k, i)
      si fila no está en lista entonces
            lista ← adicionar(fila)
            comodinesEnFila ← 0
```

```
            para j ← 0 hasta k Con paso 1 hacer
                  si filaj = comodin entonces
                        comodinesEnFila ← comodinesEnFila+1
                  fin si
            fin para
            indicesSegunComodin comodinesEnFila ← adicionar(i)
      fin si
fin para
C ← reconstruirMatriz(indicesSegunComodin)
```

Figura 10: *Algoritmo para borrar filas repetidas y ordenarla por la cantidad de comodines.*

```
metodoCombinarFilas:
para p ← 0 hasta n con paso 1 hacer
      filaP ← filaPenC
      si filaP marcada entonces continúe fin si
      para q ← 0 hasta n con paso 1 hacer
            filaQ ← filaQenC
            si filaQ marcada entonces continúe fin si
                  Combinación ← combinar(filaP,filaQ)
                  si Combinación ≠ Ø entonces
                        listaCombinan ← adicionar (Combinación)
                  fin si
      fin para
      si listaCombinan ≠ Ø entonces
            Matriz_C ←  incluir(filaP)
      sino entonces
            mejor ← mejorCombinacion(listaCombinan)
            filaY ← filaCombinadaConP
            Matriz_C ← adicionar(mejor)
            filaY ← marcar
      fin si
fin para
C ←  Matriz_C
```

Figura 11: *Algoritmo para combinación de filas*

Ejemplo del proceso de Post-Optimización:

Reducción de un CA con alfabeto 6^6 a uno de alfabeto 2^6 utilizando el proceso de Post-Optimización previamente presentado. A continuación se describe el proceso de reducción en dos partes, el primero comprende la adecuación del CA original. El segundo comprende la etapa repetitiva para la búsqueda de comodines y combinación de filas, en caso de ser necesario. Al final de cada parte se podrá observar con detalles el proceso seguido a través de las figuras correspondientes.

4.2.1 Adecuación del CA original al deseado

1. Introducción del CA en una matriz C de N*K, donde N=47 y K=6 (**ver Figura 12 a**).
2. Pasar la matriz C del alfabeto original al alfabeto objetivo (en este caso 2^6), por lo que los números permitidos en cada columna, son solamente 0 y 1, todos los números mayores que 1 se representan con -1, para facilitar la lectura, se muestran con "-" (**ver Figura 12 b**).

3. Borrar las filas que tienen más de K - t (6 – 2 = 4) comodines. Las filas que se borraron estaban ubicadas en la figura anterior en las líneas 6, 9, 10, 11, 14, 15, 16, 21, 26, 33, 34, 35, 38, 39, 40 y 43 (**ver Figura 12 c**).
4. Borrar las filas repetidas y ordenar la matriz dejando al final las que tienen mayor número de comodines. La fila que se borró en la figura anterior estaba en el renglón 29, y ahora está representada por el renglón 7 (**ver Figura 12 d**).

1	3	0	1	4	0	5
2	5	0	2	5	1	0
3	4	0	0	2	2	2
4	1	0	4	1	3	3
5	0	0	3	0	4	1
6	2	0	2	3	5	4
7	2	1	2	1	0	0
8	5	1	4	0	2	1
9	5	1	5	4	3	2
10	3	1	4	2	4	4
11	4	1	3	5	5	3
12	4	2	4	3	0	1
13	3	2	0	4	1	4
14	2	2	3	4	2	3
15	0	2	2	2	3	2
16	2	2	5	5	4	5
17	1	2	1	0	5	0
18	0	3	0	3	0	3
19	1	3	3	2	1	5
20	0	3	4	5	2	0
21	4	3	5	0	3	4
22	1	3	2	4	4	1
23	3	3	1	1	5	2
24	5	4	3	1	0	4
25	3	4	2	0	1	3
26	3	4	5	3	2	5
27	3	4	1	5	3	1
28	4	4	0	4	4	0
29	2	4	0	2	5	1
30	1	5	0	0	0	2
31	0	5	5	1	1	1
32	1	5	1	5	2	4
33	3	5	3	3	3	0
34	5	5	1	2	4	3
35	0	5	4	4	5	5
36	4	1	0	1	3	5
37	5	3	0	5	0	2
38	5	2	2	1	2	5
39	4	5	2	3	4	6
40	5	0	5	3	5	3
41	0	1	3	1	4	2
42	0	4	1	6	6	4
43	2	3	4	0	3	5
44	1	4	4	3	1	2

1	-	0	1	-	0	-
2	-	0	-	-	1	0
3	-	0	0	-	-	-
4	1	0	-	1	-	-
5	0	0	-	0	-	1
6	-	0	-	-	-	-
7	-	1	-	1	0	0
8	-	1	-	0	-	1
9	-	1	-	-	-	-
10	-	1	-	-	-	-
11	-	1	-	-	-	-
12	-	-	-	-	0	1
13	-	-	0	-	1	-
14	-	-	-	-	-	-
15	0	-	-	-	-	-
16	-	-	-	-	-	-
17	1	-	1	0	-	0
18	0	-	0	-	0	-
19	1	-	-	-	1	-
20	0	-	-	-	-	0
21	-	-	-	0	-	-
22	1	-	-	-	-	1
23	-	-	1	1	-	-
24	-	-	-	1	0	-
25	-	-	-	0	1	-
26	-	-	-	-	-	-
27	-	-	1	-	-	1
28	-	-	0	-	-	0
29	-	-	0	-	-	1
30	1	-	0	0	0	-
31	0	-	-	1	1	1
32	1	-	1	-	-	-
33	-	-	-	-	-	0
34	-	-	1	-	-	-
35	0	-	-	-	-	-
36	-	1	0	1	-	-
37	-	-	0	-	0	-
38	-	-	-	1	-	-
39	-	-	-	-	-	0
40	-	0	-	-	-	-
41	0	1	-	1	-	-
42	0	-	1	0	0	-
43	-	-	-	0	-	-
44	1	-	-	-	1	-

1	-	0	1	-	0	-
2	-	0	-	-	1	0
3	-	0	0	-	-	-
4	1	0	-	1	-	-
5	0	0	-	0	-	1
6	-	1	-	1	0	0
7	-	1	-	0	-	1
8	-	-	-	-	0	1
9	-	-	0	-	1	-
10	1	-	1	0	-	0
11	0	-	0	-	0	-
12	1	-	-	-	1	-
13	0	-	-	-	-	0
14	1	-	-	-	-	1
15	-	-	1	1	-	-
16	-	-	-	1	0	-
17	-	-	-	0	1	-
18	-	-	1	-	-	1
19	-	-	0	-	-	0
20	-	-	0	-	-	1
21	1	-	0	0	0	-
22	0	-	-	1	1	1
23	1	-	1	-	-	-
24	-	1	0	1	-	-
25	-	-	0	-	0	-
26	0	1	-	1	-	-
27	0	-	1	0	0	-
28	1	-	-	-	1	-
29	1	1	-	-	0	0
30	-	-	1	0	1	-
31	-	1	1	-	1	0

c

1	0	0	-	0	-	1
2	-	1	-	1	0	0
3	1	-	1	0	-	0
4	1	-	0	0	0	-
5	0	-	-	1	1	1
6	0	-	1	0	0	-
7	1	1	-	-	0	0
8	-	0	1	-	0	-
9	-	0	-	-	1	0
10	1	0	-	1	-	-
11	-	1	-	0	-	1
12	0	-	0	-	0	-
13	-	1	0	1	-	-
14	0	1	-	1	-	-
15	-	-	1	0	1	-
16	-	1	1	-	1	-
17	-	0	0	-	-	-
18	-	-	-	-	0	1
19	-	-	0	-	1	-
20	1	-	-	-	1	-
21	0	-	-	-	-	0
22	1	-	-	-	-	1
23	-	-	1	1	-	-
24	-	-	-	1	0	-
25	-	-	-	0	1	-
26	-	-	1	-	-	1
27	-	-	0	-	-	0
28	-	-	0	-	-	1
29	1	-	1	-	-	-
30	-	-	0	-	0	-

d

45	1	1	5	2	0	0
46	2	5	1	6	1	2
47	4	1	1	3	1	6

a

45	1	1	-	-	0	0
46	-	-	1	0	1	-
47	-	1	1	-	1	0

b

Figura 12: *CA original, adaptación al alfabeto requerido, borrar filas con más de (K-t) comodines, eliminar repetidos y ordenar.*

4.2.2 Etapa repetitiva para la búsqueda de comodines y combinación de filas

Para completar el proceso de Post-Optimización de acuerdo con el ejemplo, se deben realizar ocho iteraciones de las cuales a continuación sólo se presenta la primera iteración. Esto se debe a que el proceso se repite ocho veces utilizando la misma técnica de reducción.

Primera iteración

a. Verificar que la matriz C (que representa el CA) sigue siendo un CA y en el mismo proceso re-calcular la lista de combinaciones repetidas.

Se muestra la fila de J que indica las columnas que se toman de la matriz C, así como la cardinalidad de las dos columnas en el vector V. Los valores están subrayados con color para entender mejor la operación de transformación para el conteo de las veces que ocurre cada combinación (**ver Figura 13**).

1	0	1
2	0	2
3	1	2
4	0	3
5	1	3
6	2	3
7	0	4
8	1	4
9	2	4
10	3	4
11	0	5
12	1	5
13	2	5
14	3	5
15	4	5

J

V={2,2,2,2,2,2}

1	0	0	-	0	-	1
2	-	1	-	1	0	0
3	1	-	1	0	-	0
4	1	-	0	0	0	-
5	0	-	-	1	1	1
6	0	-	1	0	0	-
7	1	1	-	-	0	0
8	-	0	1	-	0	-
9	-	0	-	-	1	0
10	1	0	-	1	-	-
11	-	1	-	0	-	1
12	0	-	0	-	0	-
13	-	1	0	1	-	-
14	0	1	-	1	-	-
15	-	-	1	0	1	-
16	-	1	1	-	1	-
17	-	0	0	-	-	-
18	-	-	-	-	0	1
19	-	-	0	-	1	-
20	1	-	-	-	1	-
21	0	-	-	-	-	0
22	1	-	-	-	-	1
23	-	-	1	1	-	-
24	-	-	-	1	0	-
25	-	-	-	0	1	-
26	-	-	1	-	-	1
27	-	-	0	-	-	0

1	0	0	0 X 2 + 0 = **0**
2	-	1	
3	1	-	
4	1	-	
5	0	-	
6	0	-	
7	1	1	1 X 2 + 1 = **3**
8	-	0	
9	-	0	
10	1	0	1 X 2 + 0 = **2**
11	-	1	
12	0	-	
13	-	1	
14	0	1	0 X 2 + 1 = **1**
15	-	-	
16	-	1	
17	-	0	
18	-	-	
19	-	-	
20	1	-	
21	0	-	
22	1	-	
23	-	-	
24	-	-	
25	-	-	
26	-	-	
27	-	-	

1	**1**	**1**	**1**	**1**
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0

P

28	-	-	0	-	-	1
29	1	-	1	-	-	-
30	-	-	0	-	0	-

C

28	-	-		
29	1	-		
30	-	-		

Transformación

Figura 13: *Representación de las filas de J, las columnas de la matriz C, el proceso de transformación y la matriz de control P.*

Se puede observar que todas las combinaciones necesarias en fuerza 2 están presentes en esta fila de P, dado que no tiene ningún cero.

P muestra la Cantidad de veces que una combinación está presente en la matriz C. Un valor cero significa que la combinación no existe, por tanto no es un CA y un valor mayor que 1, significa que la combinación esta repetida. A continuación se muestra la matriz P después de realizar el mismo proceso con todas las filas de J, la primera fila [1 1 1 1] es la obtenida en proceso anterior (**ver Figura 14 a**).

Lista de valores repetidos: Con base en las combinaciones repetidas y utilizando J y la multiplicación sintética se encuentra las combinaciones repetidas que indican que valores faltan y en cuales columnas (**ver Figura 14 b**).

b. Establecer comodines empezando por la última fila de la matriz C (y continuando hacia la primera), basándose en la lista de combinaciones repetidas. Si no es posible hacerlo en la última, se sigue buscando en la siguiente hacia arriba. Se encontró un comodín en la fila 30 (marcado con verde) (**ver Figura 14 c**).

c. Borrar las filas con más de (K - t) comodines. Se borró la fila 30 (**ver Figura 14 d**).

P

1	**1**	**1**	**1**	**1**
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0

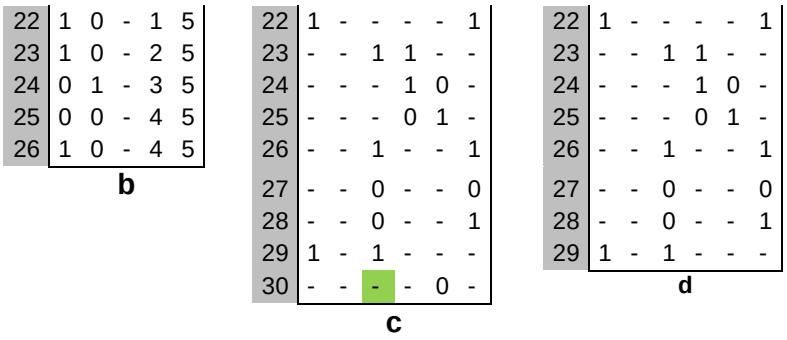
a

Faltantes

1	1	1	-	0	2	
2	0	0	-	0	3	
3	0	1	-	0	3	
4	1	0	-	0	3	
5	1	1	-	1	3	
6	1	1	-	1	3	
7	1	0	-	2	3	
8	1	0	-	2	3	
9	0	0	-	0	4	
10	1	0	-	0	4	
11	1	0	-	1	4	
12	0	0	-	2	4	
13	0	0	-	2	4	
14	1	0	-	2	4	
15	1	1	-	2	4	
16	0	0	-	3	4	
17	0	1	-	3	4	
18	1	0	-	3	4	
19	0	1	-	0	5	
20	1	0	-	0	5	
21	1	0	-	1	5	

1	0	0	-	0	-	1
2	-	1	-	1	0	0
3	1	-	1	0	-	0
4	1	-	0	0	0	-
5	0	-	-	1	1	1
6	0	-	1	0	0	-
7	1	1	-	-	0	0
8	-	0	1	-	0	-
9	-	0	-	-	1	0
10	1	0	-	1	-	-
11	-	1	-	0	-	1
12	0	-	0	-	0	-
13	-	1	0	1	-	-
14	0	1	-	1	-	-
15	-	-	1	0	1	-
16	-	1	1	-	1	-
17	-	0	0	-	-	-
18	-	-	-	-	0	1
19	-	-	0	-	1	-
20	1	-	-	-	1	-
21	0	-	-	-	-	0

1	0	0	-	0	-	1
2	-	1	-	1	0	0
3	1	-	1	0	-	0
4	1	-	0	0	0	-
5	0	-	-	1	1	1
6	0	-	1	0	0	-
7	1	1	-	-	0	0
8	-	0	1	-	0	-
9	-	0	-	-	1	0
10	1	0	-	1	-	-
11	-	1	-	0	-	1
12	0	-	0	-	0	-
13	-	1	0	1	-	-
14	0	1	-	1	-	-
15	-	-	1	0	1	-
16	-	1	1	-	1	-
17	-	0	0	-	-	-
18	-	-	-	-	0	1
19	-	-	0	-	1	-
20	1	-	-	-	1	-
21	0	-	-	-	-	0



22	1	0	-	1	5
23	1	0	-	2	5
24	0	1	-	3	5
25	0	0	-	4	5
26	1	0	-	4	5

b

22	1	-	-	-	-	1
23	-	-	1	1	-	-
24	-	-	-	1	0	-
25	-	-	-	0	1	-
26	-	-	1	-	-	1
27	-	-	0	-	-	0
28	-	-	0	-	-	1
29	1	-	1	-	-	-
30	-	-	-	-	0	-

c

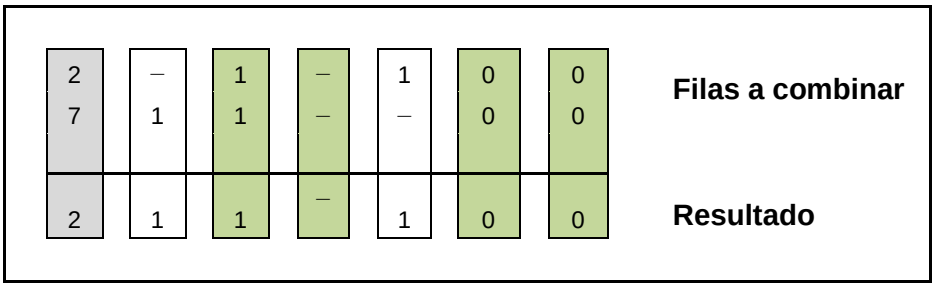
22	1	-	-	-	-	1
23	-	-	1	1	-	-
24	-	-	-	1	0	-
25	-	-	-	0	1	-
26	-	-	1	-	-	1
27	-	-	0	-	-	0
28	-	-	0	-	-	1
29	1	-	1	-	-	-

d

Figura 14: *La grafica representa la matriz de control P, combinaciones faltantes, establecer comodín y borrar fila.*

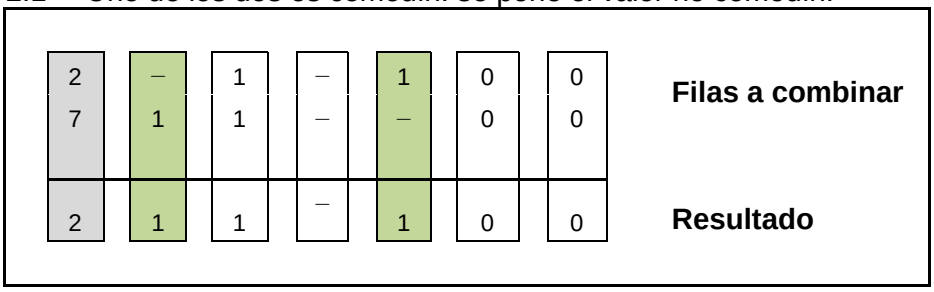
d. De la matriz anterior (**ver Figura 14 d**), tomar parejas de filas (**ver Figura 15 a**) y combinarlas si se puede, dejando como resultado una fila que representa a las dos en caso de poder combinarse (**ver Figura 15 b**).
Para ilustrar las operaciones válidas, se muestra cómo se obtuvo la fila marcada en verde, junto con las operaciones que se realizaron y las filas de la matriz anterior de donde se formaron.

Caso 1: los valores de ambas filas en la misma posición son iguales: el resultado en la fila combinada es el valor común, sea o no sea comodín.

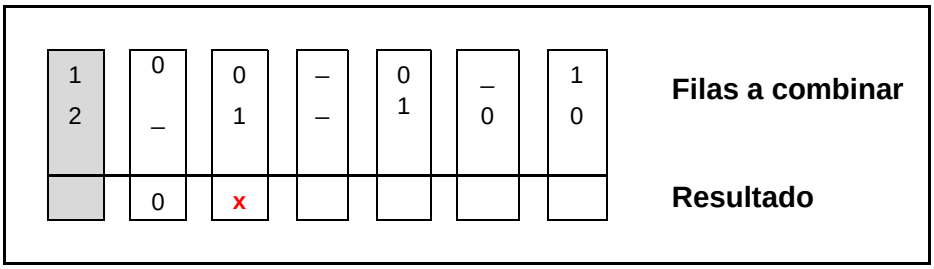


Caso 2: los valores son diferentes:

2.1 Uno de los dos es comodín: se pone el valor no comodín.



2.2 Ninguno es comodín: La combinación no se puede hacer. Para ilustrar esta operación no valida se toman otras dos filas que no se pueden combinar.



e. Borrar filas repetidas y ordenar por comodín (Ubicando las filas con mayor cantidad de comodines al final (**ver Figura 15 b**).

1	0	0	-	0	-	1
2	-	1	-	1	0	0
3	1	-	1	0	-	0
4	1	-	0	0	0	-
5	0	-	-	1	1	1
6	0	-	1	0	0	-
7	1	1	-	-	0	0
8	-	0	1	-	0	-
9	-	0	-	-	1	0
10	1	0	-	1	-	-
11	-	1	-	0	-	1
12	0	-	0	-	0	-
13	-	1	0	1	-	-
14	0	1	-	1	-	-
15	-	-	1	0	1	-
16	-	1	1	-	1	-
17	-	0	0	-	-	-
18	-	-	-	-	0	1
19	-	-	0	-	1	-
20	1	-	-	-	1	-
21	0	-	-	-	-	0
22	1	-	-	-	-	1
23	-	-	1	1	-	-
24	-	-	-	1	0	-
25	-	-	-	0	1	-
26	-	-	1	-	-	1
27	-	-	0	-	-	0
28	-	-	0	-	-	1
29	1	-	1	-	-	-

a

1	0	0	1	0	0	1
2	1	1	-	1	0	0
3	0	1	-	1	1	1
4	1	-	1	0	-	0
5	1	-	0	0	0	-
6	-	0	1	-	0	1
7	-	0	0	-	1	0
8	1	0	-	1	1	-
9	1	1	-	0	-	1
10	0	-	0	-	0	0
11	-	1	0	1	1	-
12	-	1	1	1	1	-
13	-	-	1	1	0	1
14	-	-	1	0	1	-
15	-	-	0	-	-	1

b

Figura 15: *Combinación de parejas de filas y ordenamiento.*

En el numeral **4.3** y **4.4** se describe con detalle el proceso de validación de un CA y un MCA respectivamente.

4.3 Optimización a través de la meta-heurística de Recocido Simulado

Terminado el proceso de Post-Optimización es preciso mencionar que la reducción realizada al CA todavía puede ser mayor. Es decir se puede obtener un CA o MCA de menor cantidad de filas. Para esto se procede con el uso del algoritmo Recocido Simulado.

4.3.1 Recocido Simulado

El recocido simulado o cristalización simulada, es un algoritmo meta-heurístico usado en problemas de optimización. Su principal virtud consiste en el hallazgo de la mejor solución a un problema complejo entre un conjunto de soluciones posibles con restricciones de tiempo de ejecución.

El recocido simulado surge de una analogía entre el proceso de calentamiento y enfriamiento de metales. El metal es sometido a grandes temperaturas hasta conseguir que sus partículas logren su estado de máxima energía y puedan desplazarse de su estado inicial permitiendo así un estado líquido. Posteriormente la temperatura se reduce lentamente de manera que las partículas del metal se reacomoden en estados de baja energía hasta conseguir un sólido con sus partículas acomodadas conforme a una estructura de cristal muy resistente [64].

Para la implementación computacional de esta técnica se requiere de tres parámetros para la programación de su temperatura; el parámetro para su temperatura inicial *T0*, la temperatura final *Tf,* y el grado de enfriamiento α (0 < α < 1). Para iniciar con el proceso del recocido simulado, la temperatura *Tf* se establece como *T0*, posteriormente en el ciclo de repetición se asigna *αTt* a cada iteración *Tt+1* hasta lograr el valor de *Tf.* Siguiendo con la analogía, los cambios de estado del metal en el algoritmo se presentan como Si con energía *Ei* y cada cambio de estado se produce por medio de perturbaciones, representados como *Sj* con energía Ej. Si *Sj* con energía *Ej* es menor que Si con energía *Ei*, se establece un nuevo estado, de lo contrario *Sj* es sometido a una probabilidad para poder ser aceptado ($\text{probabilidad} = e^{-(\frac{Ej-Ei}{Kb*Tt})}$, 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 *Tt.*

Para lograr que el algoritmo no quede en un ciclo infinito, se 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 [64].

En esta investigación se realiza una adecuación del método de recocido simulado [23] junto con dos funciones de vecindad que a la vez permiten el hallazgo de nuevas y mejores soluciones para los CAs. Las funciones utilizadas son *NF1* y *NF2*. Donde *NF1* recoge una t-tupla faltante al azar e intenta establecerla en la primera fila elegida de manera aleatoria en la matriz C*,* permitiendo mejorar la solución actual y *NF2* selecciona al azar *2t* celdas de la matriz *C* y trata todos los posibles cambios de símbolo de las celdas seleccionadas, estableciendo sólo 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 de *C.*

En el proceso de búsqueda de la mejor solución, se aplicó una combinación de las funciones de vecindad *NF1* y *NF2*: *NF1* se utilizó con una probabilidad p y *NF2* con una probabilidad *(1 – p).* Donde *p* varía entre 0.55 y 0.85*.*

Para el periodo de enfriamiento se definen los parámetros mencionados en la descripción del recocido simulado; temperatura inicial (T_0); temperatura final (T_f); y coeficiente de enfriamiento *(α).* Además, se añade dos parámetros, el primero sirve para medir el número de decrementos de temperatura llamada factor congelado *(FF)* que funciona como criterio de terminación alternativo cuando la búsqueda se ha estancado.

De acuerdo con la literatura encontrada [23], se determina que para un óptimo funcionamiento del recocido simulado, las variables se definen con un valor inicial de; a) *T0*=1; b) *α*=0,99; c) $L=Nkv^2$, d) *Tf = 0,0000000001*; e) *FF = 33*; y f) *k'max = t + 1.*

En la **Figura 16** se presenta en pseudocódigo la adecuación del algoritmo entre el recocido simulado y las funciones de vecindad mencionadas.

```
RecocidoSimulado: 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 ← getTamanioFaltantes(),evalActual ← evalC+1
para i ← 1 hasta 2 Con paso 1 hacer
mientras T>Tf y f(Ca,n,k,v,t)>0 y FC<FF y evalActual!=0 hacer
            para j ← 0 hasta L y evalActual!=0 Con paso 1 hacer
                  rand ← random()
                  si rand < p
                        C''← NF1(C', n, k, v, t, k', evalActual)
                  fin si
                  sino
                        C''← NF2(C', v, evalActual, n, k, t, k')
                  fin sino
                  min ← minimo(evalActual, evalC, temp)
                  si rand < min o evalActual < evalC
                        C ← copiarC(C'', n, k)
                        evalC ← evalActual
                        k' ← 0
                  fin si
                  sino
                        k'← k'+1
                  fin sino
                  si f(C, n, k, v, t) < f(Ca, n, k, v, t)
                        Ca ← copiarC(C, n, k)
```

```
                        FC ← 0
                    fin si
                    si k'< kmax
                        k'← 0
                    fin si
                fin para
                T ← T * α
                FC ← FC + 1
        fin mientras
fin para
cRecibido ← Ca
si TestCA(Ca, v, n, k, t)
        salida ← verdadero
fin si
sino
salida ← falso
fin sino
fin funcion
```

***Figura 16**: Adecuación del algoritmo de recocido simulado y funciones de vecindad.*

4.4 Proceso de validación de un CA

Un CA no necesariamente debe estar balanceado, es decir, en cada CA con una cobertura de fuerza t, se debe presentar que cada posible combinación de valores (*t-adas)* esté presente al menos una vez, pero no precisamente el mismo número de veces.

Un CA se representa como una matriz C de tamaño *N x k*, donde cada elemento c_{ij} toma como valor un símbolo perteneciente al conjunto S = {0, 1, 2, 3,…, (v-1)}, tal que cada *N x k* sub-arreglo contiene al menos una vez todas las posibles combinaciones de los v^t símbolos para alfabetos uniformes y $\prod_{i=0}^{i=k-1} vi$ para alfabetos mixtos. El número de sub-arreglos Λ de tamaño N x t está determinado por la cantidad de t-adas necesaria para cumplir con las características de un CA.

Para la comprobación de todas las posibles combinaciones de los $\prod_{i=0}^{i=k-1} vi$ símbolos aparecen al menos una vez por cada una de las Λ *t-adas* en el CA, se hace uso de un vector *V* de tamaño *k* con las cardinalidades de las variables, una matriz *P* de tamaño $\Lambda \times \prod_{i=0}^{i=t-1} maxVi$, donde $maxV_i$ contiene las t-cardinalidades de mayor valor y una matriz *J* de tamaño Λ x *t* donde cada renglón representa una *t-ada* del CA.

En la matriz *P* se mantiene el registro de las combinaciones de símbolos existentes por cada *t-ada* del CA. Cada una de las Λ *t-adas* (fila en *J*) se mapea a un renglón, en *P* en la misma fila que en J.

Para verificar que todas las combinaciones de símbolos de cada t columnas de **C** aparezcan al menos una vez, es necesario mapear cada **t** combinaciones de símbolos a un valor decimal que corresponde a una columna en P. Para ello, al vector V se le puede ver como un sistema de numeración donde cada uno de sus elementos representa el valor máximo (o la base del sistema de numeración) que puede tomar este elemento, por lo tanto, para contar los elementos de una *t-ada*, se realiza la transformación de los *t*

símbolos correspondientes a la *t-ada* representada por J_i, a un valor decimal, siguiendo la expresión 4.1.

$$\sum_{i=1}^{t-1} Ji,\ J_i = J_{(i-1)} \times V_{Ji} + J_i \quad (4.1)$$

Para alfabetos uniformes se validará que es un CA, si al final no existe ningún cero en P. Para alfabetos mixtos se validará que es un MCA, si para cada *t-ada* (J_i) no existe ningún cero en las primeras $\prod_{i=0}^{i=t-1} VJi$ columnas de su correspondiente renglón en *P*.

A continuación se expone un ejemplo para mejorar el entendimiento acerca del proceso de validación.

4.5 Proceso de validación de un MCA

Tomando como base de ejemplo el siguiente MCA (6; 3, 2^2, 3^1 ,2), donde N = 6 representa el número de filas, k = 3 el número de columnas, v = $2^{2,}3^1$ el alfabeto y t = 2 la fuerza, representado en la **Tabla 5**. Para verificar que se está en presencia de un MCA se procede de la siguiente forma: se inicializa el vector de cardinalidades V, el vector *maxV* con las *t* cardinalidades de mayor valor, la matriz de *t-adas J* y la matriz de control *P.* Para este ejemplo, V es de tamaño *k*=3, *max*V de tamaño t=2, *J* de tamaño 3 x (*t=2)* y *P* de tamaño 3 x 6, donde 3 es la cantidad de filas de *J* y 6 es el resultado de la productoría de los elementos de *max*V como se muestra en la **Tabla 6**.

1	0	0
0	1	1
0	0	2
1	1	2
1	0	1
0	1	0

Tabla 5: *MCA (6; 3, 2^2, 3^1, 2).*

V={2,2,3}

*max*V={2,3}

0	1
0	2
1	2

J

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

P

Tabla 6: *Inicialización de variables.*

A continuación se verifica que todas las combinaciones de símbolos para cada una de las *t-adas* exista. Para entenderlo mejor, se toma la primera *t-ada* de la matriz *J,* se mapea cada *t-ada* (fila en J) directamente la misma fila de P, seguidamente se leen renglón a renglón sus correspondientes símbolos en *C,* con cada *t* símbolos se obtiene el número de columnas al que corresponden en *P* de acuerdo con la expresión 4.1. Este mismo proceso se efectúa por cada *t-ada*. El proceso completo se muestra como se sigue en las tablas 6, 7, 8, 9.

0	1		1	0	0		1	0	=	1	x	2	+	0	=	**2**	**1**	**2**	**1**	0	0
0	2		0	1	1		0	1	=	0	x	2	+	1	=	**1**	0	0	0	0	0
1	2		0	0	2		0	0	=	0	x	2	+	0	=	**0**	0	0	0	0	0
J			1	1	2		1	1	=	1	x	2	+	1	=	**3**	P				
			1	0	1		1	0	=	1	x	2	+	0	=	**2**					
V={2,2,3}			0	1	0		0	1	=	0	x	2	+	1	=	**1**					
			C				Columnas en Expresión 4.1														

Tabla 7: *Validando un MCA, t-ada {0,1}.*

0	1		1	0	0		1	0	=	1	x	3	+	0	=	**3**	1	2	2	1	0	0
0	2		0	1	1		0	1	=	0	x	3	+	1	=	**1**	**1**	**1**	**1**	**1**	**1**	**1**
1	2		0	0	2		0	2	=	0	x	3	+	2	=	**2**	0	0	0	0	0	0
			1	1	2		1	2	=	1	x	3	+	2	=	**5**	P					
			1	0	1		1	1	=	1	x	3	+	1	=	**4**						
V={2,2,3}			0	1	0		0	0	=	0	x	3	+	0	=	**0**						
			C				Columnas en Expresión 4.1															

Tabla 8: *Validando un MCA,* t-ada *{0,2}.*

0	1		1	0	0		0	0	=	0	x	3	+	0	=	**0**	1	2	2	1	0	0
0	2		0	1	1		1	1	=	1	x	3	+	1	=	**4**	1	1	1	1	1	1
1	2		0	0	2		0	2	=	0	x	3	+	2	=	**2**	**1**	**1**	**1**	**1**	**1**	**1**
			1	1	2		1	2	=	1	x	3	+	2	=	**5**	P					
			1	0	1		0	1	=	0	x	3	+	1	=	**1**						
V={2,2,3}			0	1	0		1	0	=	1	x	3	+	0	=	**3**						
			C				Columnas en Expresión 4.1															

Tabla 9: *Validando un MCA,* t-ada *{1,2}.*

Una vez terminado el proceso se puede asegurar que el array evidenciado en la **Tabla 5** es un MCA valido. Debido a que para cada *t-ada* no existe ningún cero en las primeras $\prod_{i=0}^{i=t-1} VJi$ columnas de su correspondiente renglón en *P* (***ver*** **Tabla 10**).

0	1		1	0	0		0	1	=	2	x	2	=	4	**1**	**2**	**2**	**1**	0	0
0	2		0	1	1		0	2	=	2	x	3	=	6	**1**	**1**	**1**	**1**	**1**	**1**
1	2		0	0	2		1	2	=	2	x	3	=	6	**1**	**1**	**1**	**1**	**1**	**1**
J			1	1	2		*max*V								P					
			1	0	1															
V={2,2,3}			0	1	0															
			C																	

Tabla 10: *Aceptación del MCA.*

5 EXPERIMENTACIÓN Y RESULTADOS

En esta sección se detalla los resultados obtenidos en la adaptación de los CA y MCA utilizados por el complemento desarrollado y comparado con algunos de los mejores existentes en el repositorio del CINVESTAV. La etapa de experimentación 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 perteneciente a la incubadora de empresas ParqueSoft Popayán.

5.1 Resultados obtenidos mediante el método implementado

En la **Tabla 11** se muestran la comparación de los Mixed Covering Arrays obtenidos a partir del Post-Optimizador (**P**) y el algoritmo desarrollado para poblar el repositorio (Post-Optimizador y SA (**P+SA**)) contra los mejores reportados por el CINVESTAV.

Id	MCA	t	CINVESTAV	P	P+SA	Diferencia con P	Diferencia con P+SA	%P	%P+SA	P+SA (ms)	P (ms)
1	3^3 2^3	2	9	10	9	1	0	0	11,11	46809	69
2	6^1 5^5	2	30	43	34	13	4	13,33	43,33	9299600	571
3	6^4 5^2	2	37	46	45	9	8	21,62	24,32	5595818	845
4	6^5 5^1	2	39	48	46	9	7	17,95	23,08	5786827	689
5	5^1 3^4	3	45	57	45	12	0	0	26,67	227183	225
6	4^1 3^4	2	12	14	12	2	0	0	16,67	107006	96
7	4^1 2^8	2	8	10	8	2	0	0	25	176046	164
8	4^3 3^5	2	16	22	16	6	0	0	37,5	1849870	467
9	4^5 3^4	2	19	25	19	6	0	0	31,58	5372150	421
10	4^6 3^3	2	20	25	20	5	0	0	25	6462730	450
11	4^6 3^4	2	20	25	22	5	2	10	25	8265629	575
12	5^1 4^5	2	20	28	20	8	0	0	40	1877372	373
Total			275	315	296	40	21	7,63	14,55	3755586,67	412,08
										Tiempo Promedio(ms)	

Tabla 11: *Comparación de resultados entre los MCA reportados en el CINVESTAV y los adaptados con el algoritmo de Post-Optimización propuesto en esta investigación.*

De acuerdo con la información de la **Tabla 11**, se puede decir que los MCA reportados por el CINVESTAV son mejores que los obtenidos con el método de Post-Optimización propuesto en esta investigación en un porcentaje de 7,63%. Esta diferencia podría ser causada debido a que los CA y MCA del CINVESTAV fueron creados desde cero usando un clúster con las siguientes características[15]; 11032GB de RAM, 4244 Cores CPU, Disco Duro de 54TB y sistema operativo Linux. Mientras que los utilizados en esta investigación se generaron a partir de unos existentes usando una máquina con 4GB de RAM, un Disco

[15] http://clusterhibrido.cinvestav.mx/

Duro de 1TB, un procesador Intel(R) Core(TM) i7-3770 CPU y un sistema operativo Windows 7 de 64bits.

Los resultados obtenidos usando el algoritmo de Post-Optimización dependen del CA o MCA original. Por ejemplo para encontrar un CA de alfabeto 2^3 se partió de la configuración del CA(47,6,2^6,2). Consiguiendo un CA(6,3,2^3,2) que no es óptimo; mientras que partiendo de CA(8,4,2^{10},2) se obtuvo CA(4,3,2^3,2) óptimo. Por esta razón en el algoritmo propuesto se parte siempre del CA más cercano con menor cantidad de filas.

En este trabajo se tomó el más cercano con la menor cantidad de filas para encontrar un resultado más rápido en vez de uno mejor, debido a que se trata de una aplicación en tiempo real. A demás el objetivo no consiste en conseguir el mejor CA o MCA, sino obtener un resultado bueno en poco tiempo y con una máquina accesible para cualquier empresa.

5.2 Pruebas realizadas con estudiantes y grupo IDETI

Con el propósito de poner a prueba el prototipo funcional del complemento desarrollado, se procede con un experimento realizado en dos salas de la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca con 22 estudiantes de los últimos semestres del programa de Ingeniería de Sistemas. Esto con el fin de obtener una retroalimentación que permitiera mejorar y estabilizar aún más el prototipo antes de realizar las pruebas beta.

Para el desarrollo de la prueba se usaron los siguientes insumos:

1. Un equipo con VS.Net 2010, 2012 o 2013 con el complemento instalado y acceso a internet.
2. Un usuario y una contraseña de acceso.
3. Ejecutable escrito en C# que contiene una clase con los métodos a probar.
4. Archivos de entrada necesarios para probar uno de los métodos.
5. Una librería para serializar objetos necesario en el proyecto de pruebas.
6. Un archivo con la url de la encuesta.
7. Video tutorial con todos los pasos para el desarrollo de una prueba.
    a. Iniciar sesión.
    b. Cargar el ejecutable.
    c. Seleccionar el método a probar.
    d. Configurar parámetros de entrada.
    e. Generar casos de prueba.
    f. Configurar parámetros de salida y valor esperado.
    g. Crear y configurar un proyecto de pruebas.
    h. Copiar código fuente generado por el complemento, pegarlo sobre el proyecto de pruebas y ejecutar la prueba.

La prueba se planteó de la siguiente manera:

1. Se presentó una breve introducción de lo que se iba a realizar en la prueba y se entregaron todos los insumos.

2. Los testers realizaron la primera prueba utilizando el video como guía de aprendizaje para interiorizar todos los pasos.
3. Se repitió el proceso aprendido sobre los métodos de la clase contenidos en el ejecutable.
4. Se realizó la encuesta utilizando la url proporcionada en el archivo.

5.2.1 Experimento 1 - pruebas alfa con estudiantes

Este primer experimento se ve enmarcado dentro de la prueba alfa, en la cual se entregaron los insumos necesarios y se realizó como se planteó en el numeral 5.2 y se desarrolló en presencia de los desarrolladores del complemento.

En la **Figura 17** se muestra la descripción correspondiente con cada número de las características evaluadas en la encuesta realizada a los participantes, las cuales servirán de ayuda para entender de mejor manera la gráfica (**ver Figura 18**) generada de acuerdo con los datos obtenidos.

N°	Pregunta
1	Grado de satisfacción en la realización de la prueba
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

Figura 17: *Características evaluadas.*

Nomenclatura de colores					
Favorable		Intermedio	No sabe	Desfavorable	

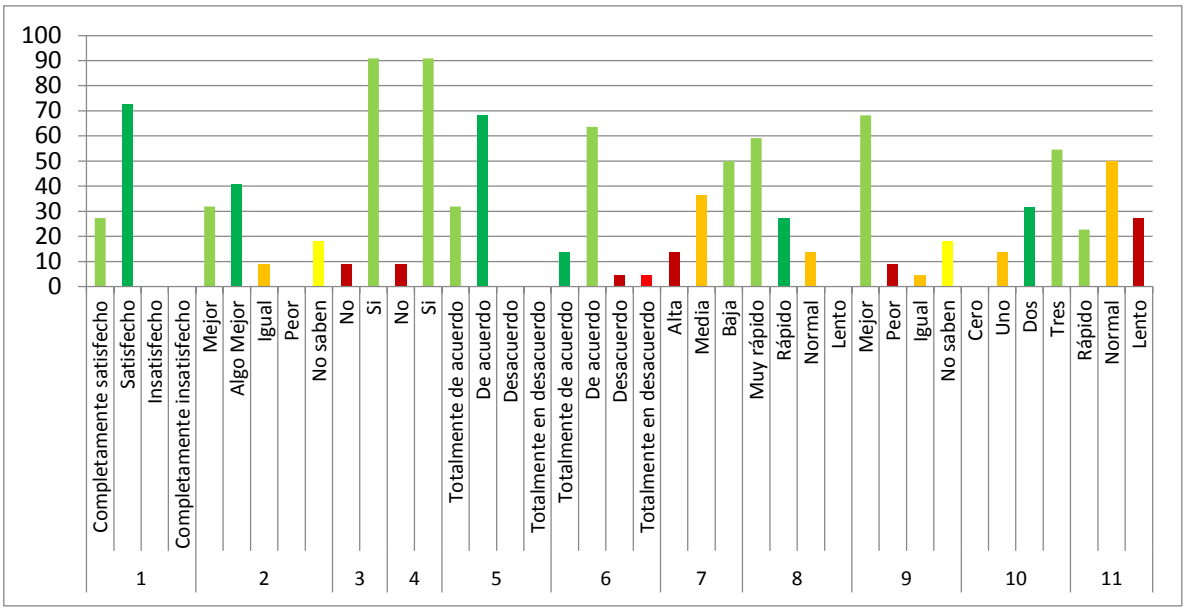


Figura 18: *Gráfica que representa la información obtenida con los 22 estudiantes en la prueba alfa. Para entender mejor esta gráfica, se recomienda ver Figura 17 donde se describen las características evaluadas.*

De acuerdo con la anterior gráfica, se observa que los resultados obtenidos son favorables en los once aspectos en los que se enfocó la encuesta. Sn embargo, las barras amarillas no otorgan información, las 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, por tanto se toman como oportunidades de mejora. La información necesaria para conseguirlo se obtuvo mediante preguntas abiertas en las que el encuestado expresaba libremente, lo que le gustó, lo que no le gustó y lo que cambiaría para mejorar.

Los resultados relevantes y destacados obtenidos a causa de los comentarios y aportes mencionados por los participantes de la prueba, fueron los siguientes:

1. Una vez cargado el ejecutable o dll, debería poder usarlo sin volver a buscarlo.
2. En caso de que los valores de una misma variable sean muy parecidas, debería de tener un texto base para concatenar y no repetir lo mismo con cada valor, tanto en el formulario de crear y configurar la prueba (2.1), como en el de configurar valores esperados y parámetros de salida (2.2).
3. Los datos de prueba no deberían agregarse por separado a cada parámetro como cadenas, sino mejor crear una clase que incluya todos los parámetros y el valor esperado con sus respectivos tipos de datos.

Con base en las sugerencias recibidas en las pruebas alfa se implementaron las mejoras. Y se procedió a realizar las pruebas beta con el grupo IDETI de Parquesoft Popayán.

5.2.2 Experimento 2 - pruebas beta con grupo IDETI

Esta etapa se realizó en dos pruebas de las cuales se obtuvieron los siguientes resultados:

5.2.2.1 Primera Prueba

La prueba se llevó a cabo en las instalaciones de ParqueSoft Popayán siguiendo los pasos descritos en el numeral 5.2 y se permitió a los ingenieros interactuar libremente con el complemento. Los resultados de la encuesta se muestran a continuación (**ver Figura 19**):

Nomenclatura de colores					
Favorable		Intermedio	No sabe	Desfavorable	

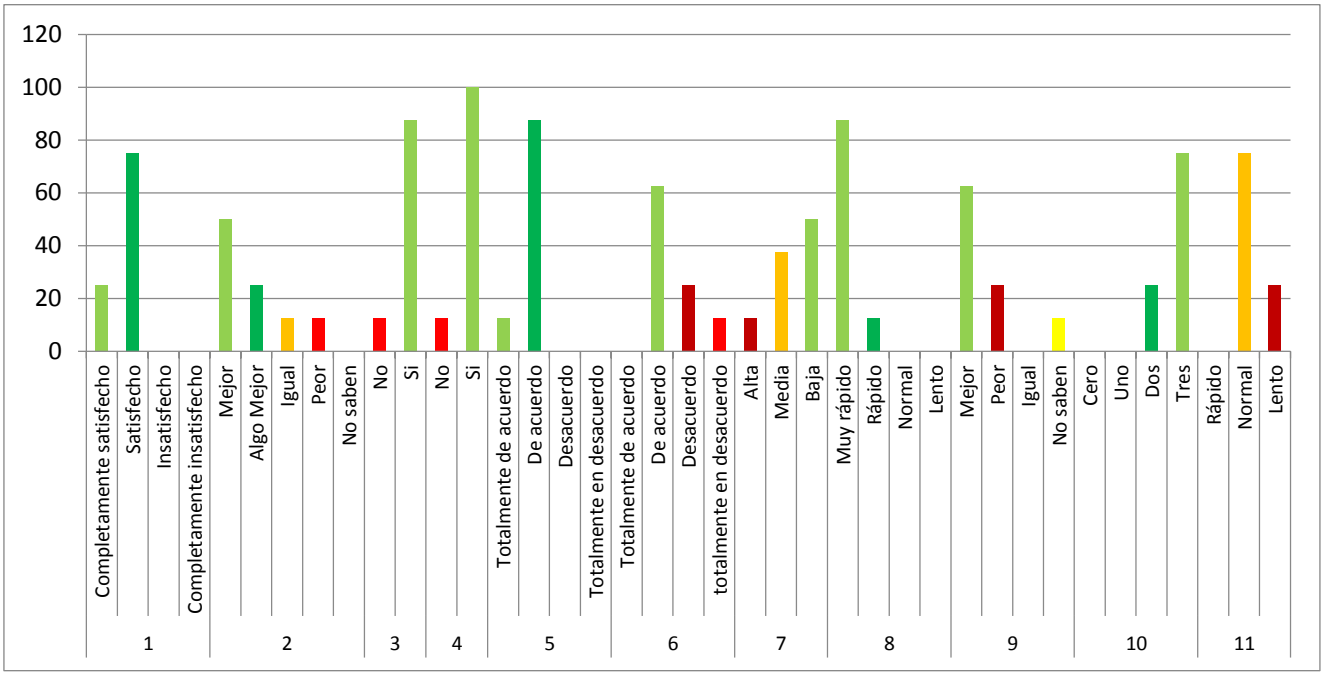


Figura 19: *Gráfica que representa la información de la encuesta realizada al grupo IDETI de ParqueSoft en la primera prueba beta. Para entender mejor esta gráfica, se recomienda ver* ***Figura 17*** *donde se describen las características evaluadas.*

De la encuesta se extrajeron las siguientes sugerencias relevantes para esta investigación:

1. Falta de información que indique para que sirve cada campo, por ejemplo el prefijo y sufijo, además la etiqueta del valor está muy lejos de la caja de texto, deberían bajar la etiqueta y subir el prefijo y sufijo.
2. En el formulario de configuración de los valores esperados y parámetros de salida se debería poder ver claramente el nombre del método, el retorno y el tipo de dato de cada variable.

3. El complemento determina si el método es correcto o no y además dice el valor el valor que se obtuvo y el que se esperaba, pero no muestra que tipo de error podría tener.

Después de implementar las dos primeras sugerencias y dejar la tercera para trabajo futuro, se procede a realizar la segunda y última prueba.

5.2.2.2 Segunda Prueba

Esta segunda y última prueba se realizó en las instalaciones de ParqueSoft Popayán. Donde por segunda vez se permitió a los ingenieros interactuar libremente con el complemento. Los resultados se muestran a continuación (**ver Figura 21**):

N°	Pregunta
1	Grado de satisfacción en la realización de la prueba
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

Figura 20: *Características evaluadas.*

Nomenclatura de colores					
Favorable		Intermedio	No sabe	Desfavorable	

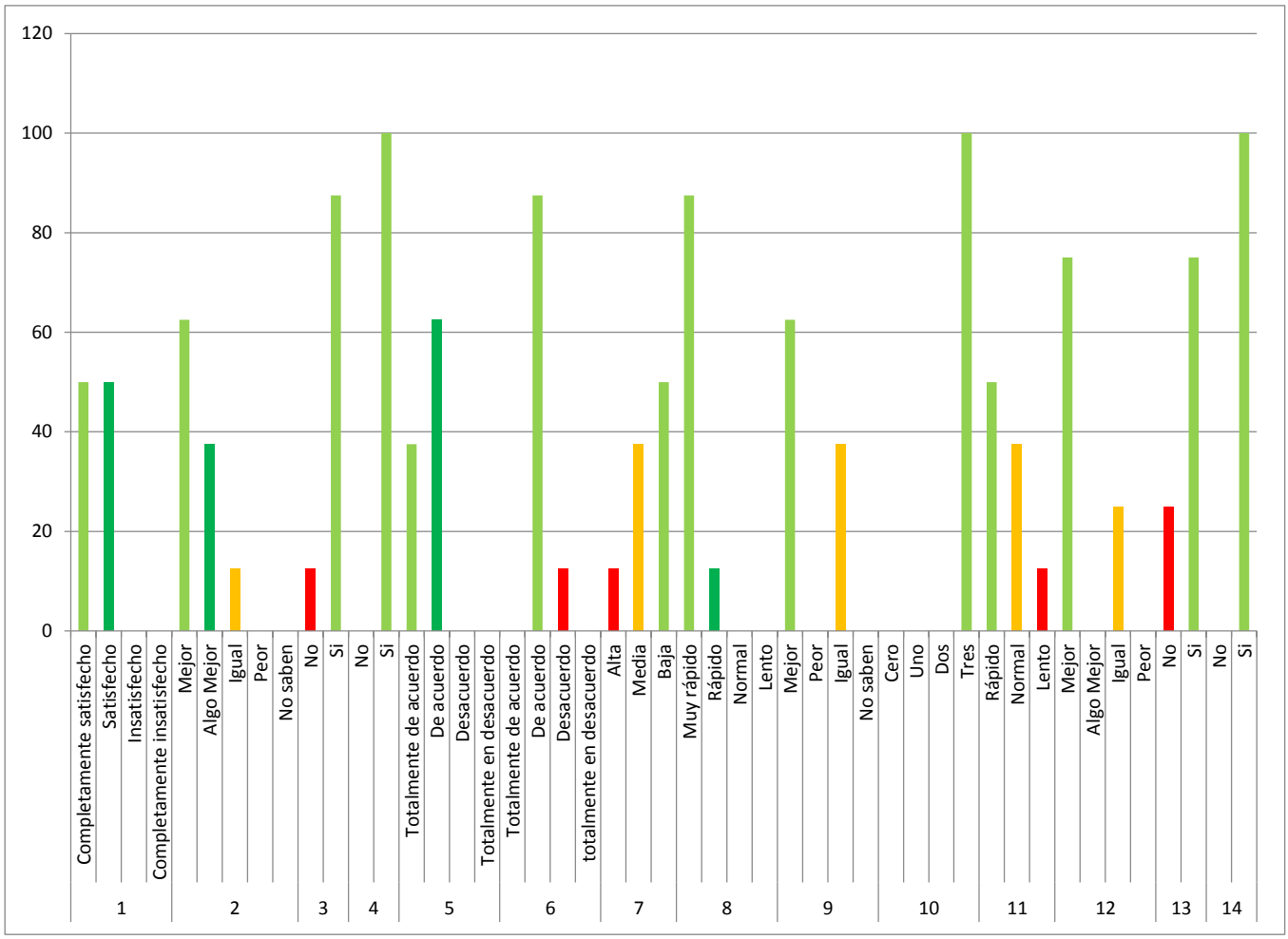


Figura 21: *Gráfica que representa la información de la encuesta realizada al grupo IDETI de ParqueSoft sobre la versión final del complemento. Para entender mejor la gráfica (ver Figura 20) donde se describen las características evaluadas.*

De acuerdo con las gráficas obtenidas en la realización de las dos pruebas, se puede decir 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 demostró conformidad con las pruebas realizadas a través del complemento desarrollado y aseguraron utilizarla en un futuro cercano sobre el desarrollo de sus proyectos software.

5.3 Versiones del complemento

En esta sección del documento se pone en evidencia la implementación de las sugerencias otorgadas por los participantes de las pruebas alfa y beta en la etapa de experimentación, las actualizaciones realizadas se pueden diferenciar sobre las imágenes de acuerdo con el color con el que se marcan los puntos de interés[16]; el recuadro de

[16] Lugar en el formulario que indica donde hay una oportunidad de mejora.

borde rojo indica que los usuarios detectaron una inconformidad en ese lugar (Antes) y un recuadro de borde verde indica como quedo el mismo formulario después de implementar la sugerencia hecha por los usuarios mediante la encuesta (Después). Las imágenes del antes y después están de forma consecutiva para poder visualizar con facilidad las diferencias.

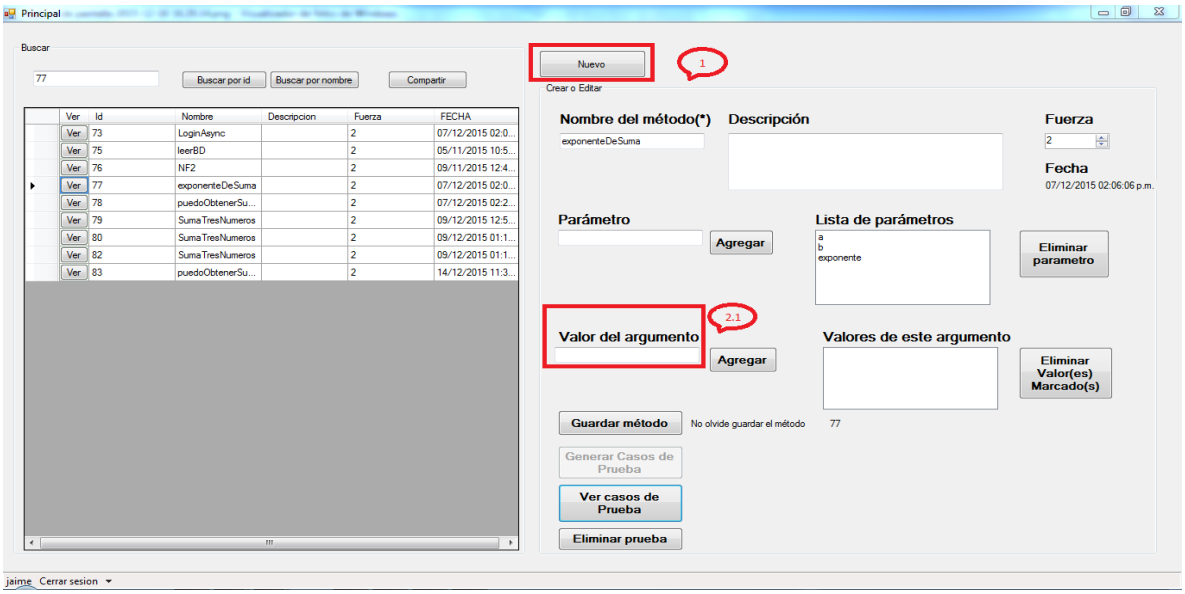


***Figura 22**: Vista del formulario principal durante la prueba alfa con estudiantes, se marcó la sugerencia 1 y 2.1.*

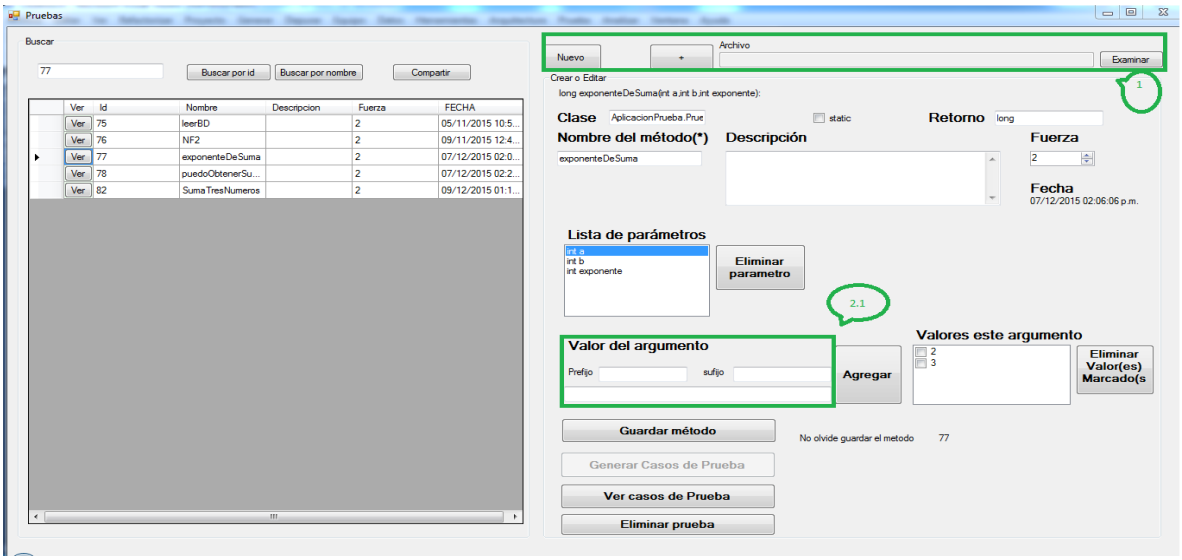


***Figura 23:** Vista del formulario principal con las sugerencia 1 y 2.1 implementadas.*

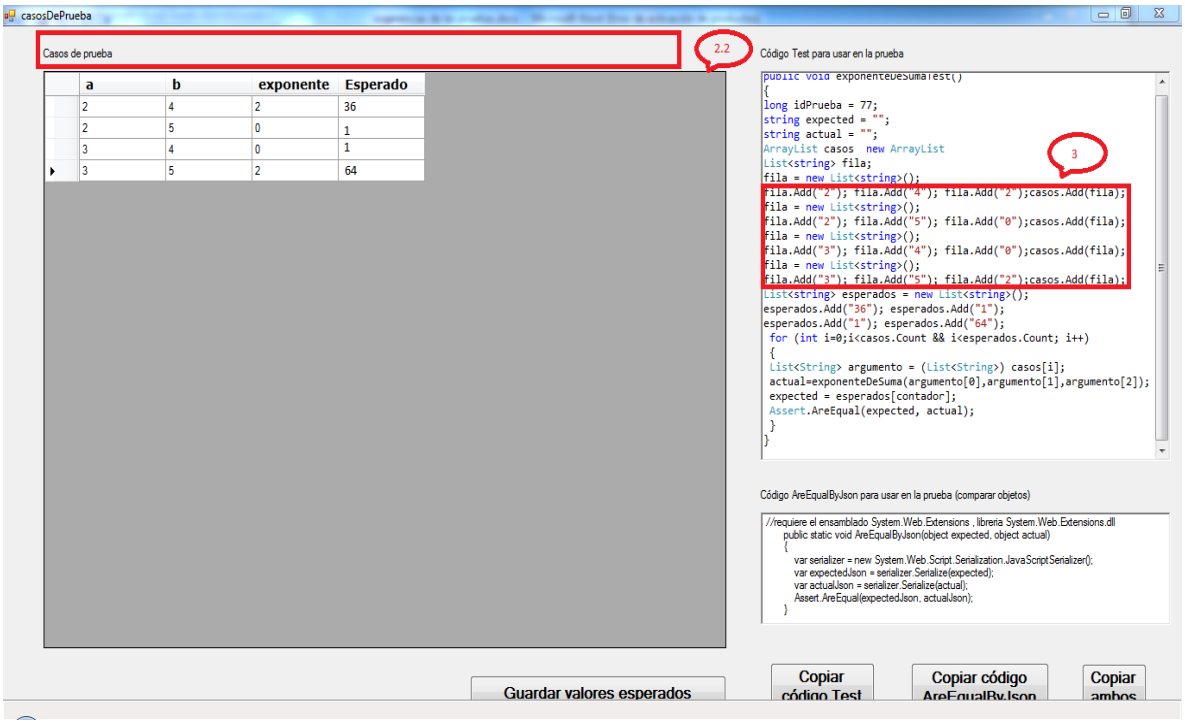


***Figura 24**: Vista del formulario de configuración de valores esperados durante la prueba alfa con estudiantes, se marcó las sugerencias 2.2 y 3.*

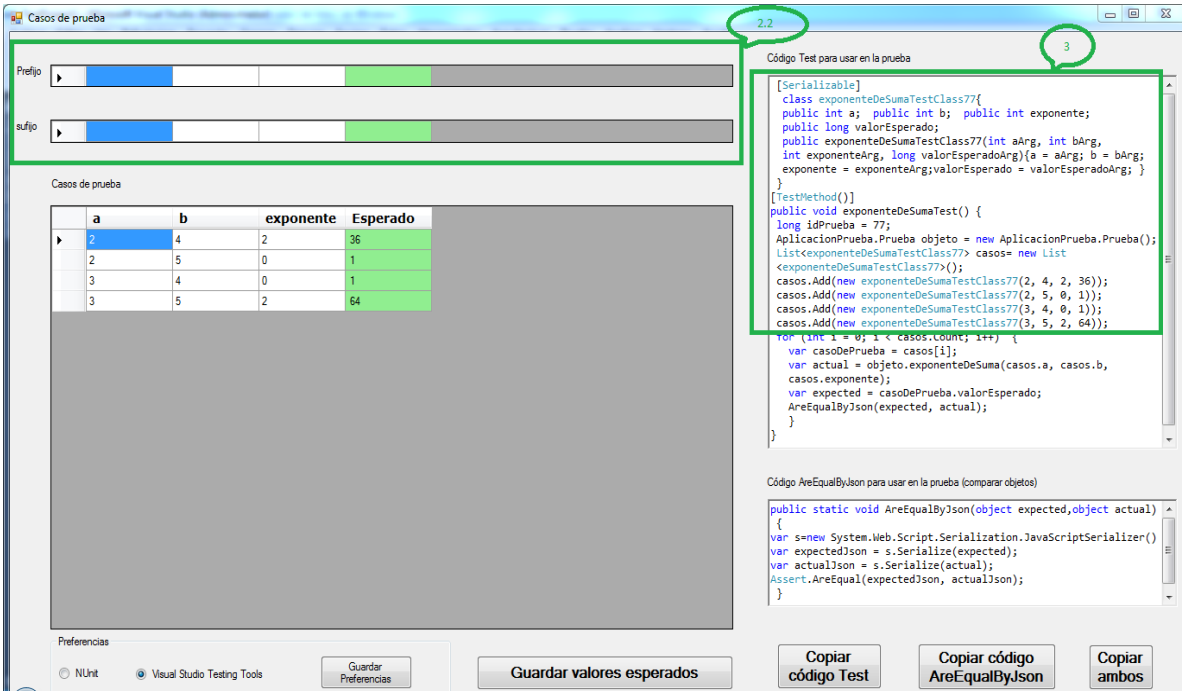


***Figura 25:** Vista del formulario de configuración de valores esperados con las sugerencias 2.2 y 3 implementadas.*

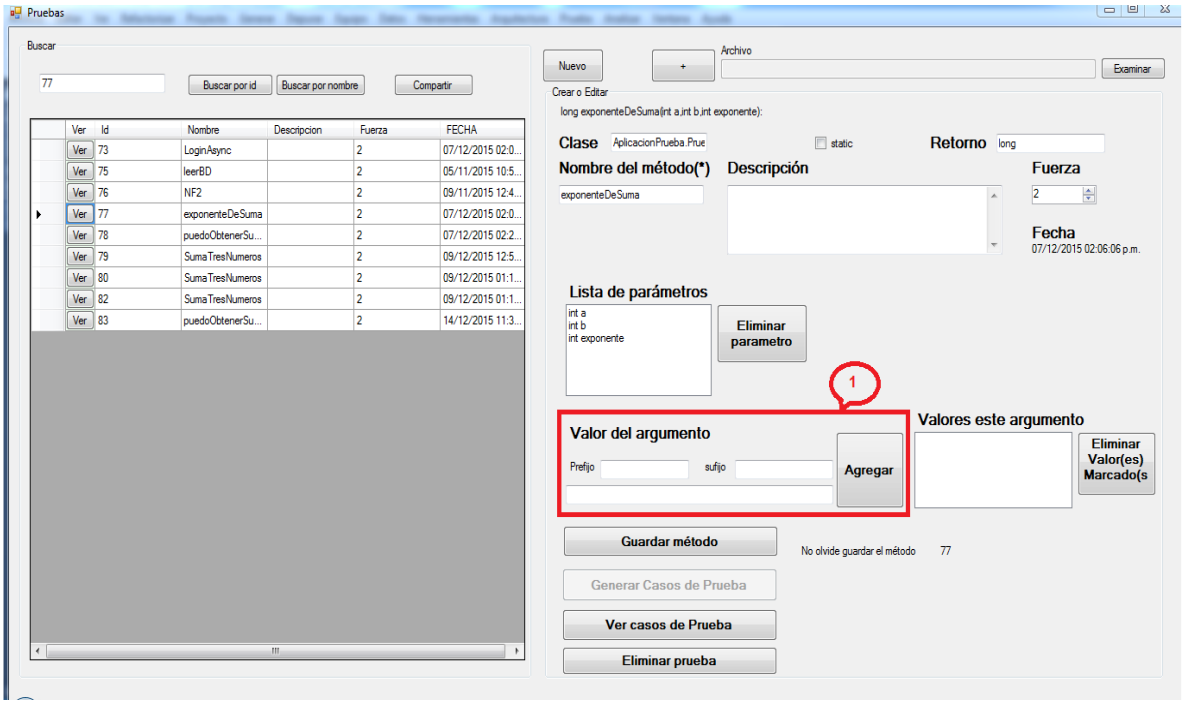


Figura 26: *Vista del formulario principal durante la primera prueba beta con ingenieros de IDETI, se marcó la sugerencia 1.*

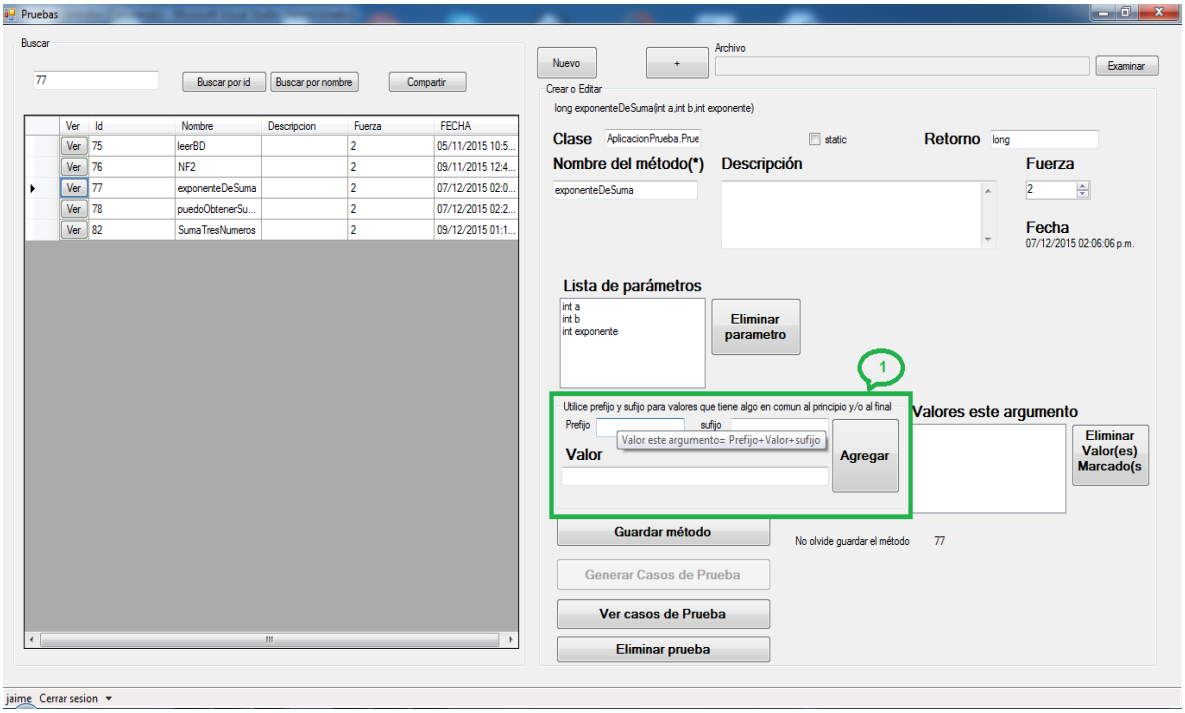


Figura 27*: Vista del formulario principal con la sugerencia 1 implementada.*

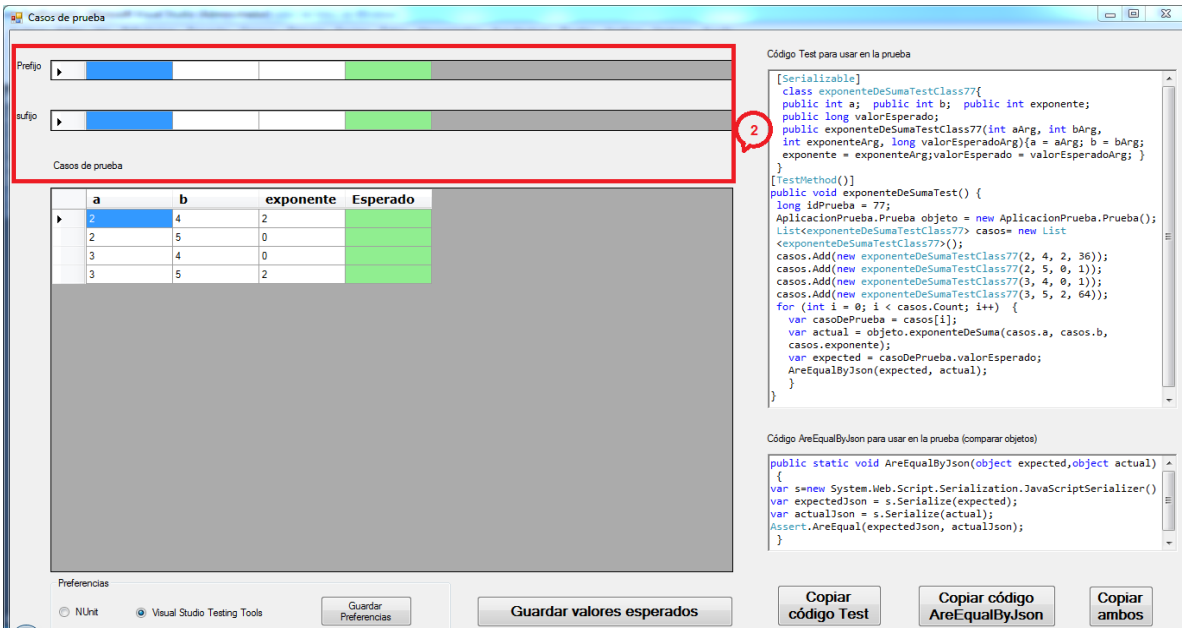


Figura 28: *Vista del formulario de configuración de valores esperados durante la primera prueba beta con los ingenieros de IDETI, Se marcó la sugerencia 2.*

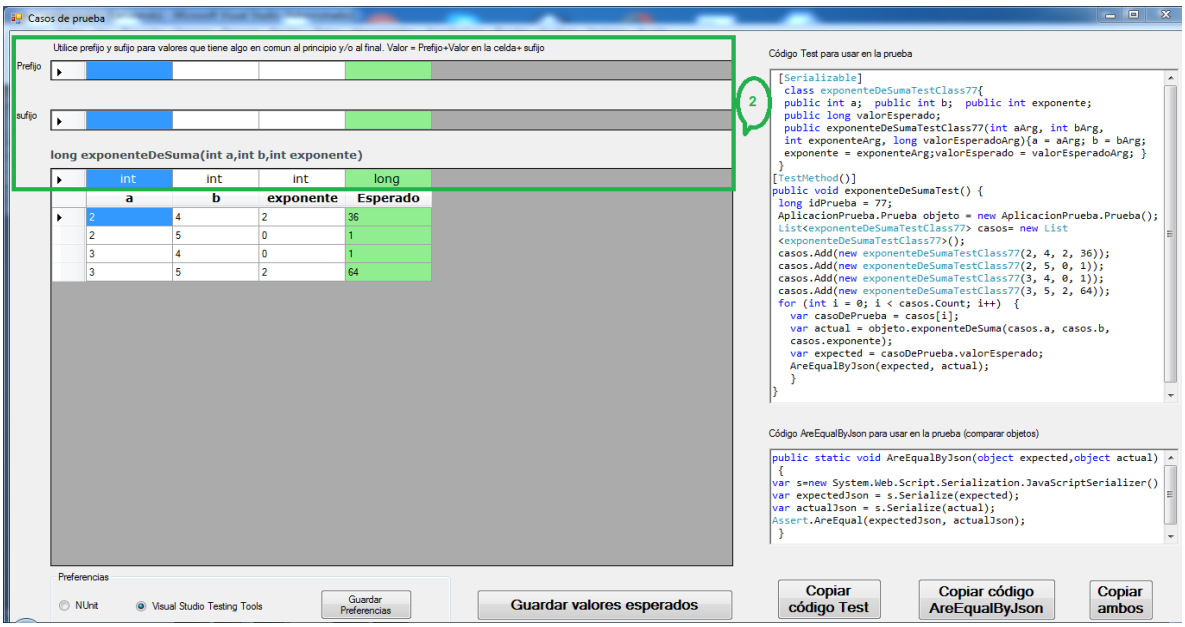


Figura 29: *Vista del formulario de configuración de valores esperados con la sugerencia 2 implementada.*

La sugerencia 3 hecha por los integrantes del grupo IDETI no se implementó, puesto que no está dentro del alcance de esta investigación, se sugiere esta modificación para trabajos futuros utilizando Error-Locating Arrays que permiten identificar y localizar los fallos [7, 65].

6 CONCLUSIONES, RECOMENDACIONES Y TRABAJO FUTURO

6.1 CONCLUSIONES

- El algoritmo de Post-Optimización propuesto para adaptar los CA o MCA del repositorio a los requerimientos de las pruebas cumplió con el objetivo y lo hizo usando en promedio 412,08 milisegundos para arreglos de menos de once columnas con fuerza dos y cardinalidades menores a siete.

- El grado de optimización del algoritmo propuesto en esta investigación en conjunto con la adaptación del Recocido simulado es aceptable, teniendo en cuenta que los resultados son 7,63% más grandes que los reportados por el CINVESTAV (**ver Tabla 11**). Utilizando una máquina con 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. Mientras que el CINVESTAV utilizó un clúster con las siguientes características[17]; 11032GB de RAM, 4244 Cores CPU, Disco Duro de 54TB y sistema operativo Linux. Resaltando además que el CINVESTAV crea los CA o MCA desde cero buscando el óptimo y no tienen limitaciones de tiempo, a diferencia del método propuesto en esta investigación el cual requiere resultados en tiempo de ejecución.

- De acuerdo con los resultados de la encuesta inicial realizada a los estudiantes y a los ingeniero del grupo IDETI en la realización de la primera prueba, se destaca que el 72.72% y 75% respectivamente de los encuestados consideran que el complemento desarrollado en comparación con otras herramientas para la realización de pruebas unitarias posee mejoras de acuerdo con el tiempo utilizado para configurar una prueba, cantidad de errores detectados, facilidad de uso y rapidez en tiempo de respuesta.

- Los resultados obtenidos de acuerdo con la última encuesta a los ingenieros del grupo IDETI una vez terminada la segunda prueba, demuestran que el complemento desarrollado logró detectar el 100% de los errores sobre el ejecutable puesto en prueba, el 75% de los encuestados destacan las mejoras realizadas sobre el primer prototipo probado y como prueba de satisfacción todo el grupo demostró estar satisfecho con la herramienta.

- La estrategia de usar el algoritmo de Post-Optimización, el generar inmediatamente la respuesta y ejecutar de forma asíncrona el algoritmo de recocido simulado dio buenos resultados, puesto que permite responder rápidamente, pero además puede encontrar un mejor CA o MCA para futuras pruebas que tengan necesidades equivalentes.

- El algoritmo de Post-Optimización propuesto utiliza solamente operaciones válidas (eliminación de filas de filas repetidas, combinación de filas, eliminación de filas que no tienen fuerza t), tales que una modificación realiza mediante la aplicación de una de ellas, 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 bueno.

[17] http://clusterhibrido.cinvestav.mx/

- Las pruebas iniciales mostraron que para algunos métodos probados, el complemento tardaba en promedio 3755586,67 milisegundos, por lo que se adoptaron dos estrategias: la primera consistió en devolver el resultado sin aplicar la adaptación del recocido simulado tardando en promedio 412,08 milisegundos; la segunda fue crear un algoritmo para generar alfabetos deseados con tres a diez parámetros y fuerza dos a seis, y se pobló el repositorio con esos alfabetos, de tal forma que al necesitar alguno de esos CA, ya no fue necesario crearlos o adaptarlos sino utilizarlos, con lo que el tiempo de respuesta alcanzó en promedio 275 milisegundos.

6.2 RECOMENDACIONES Y TRABAJO FUTURO

- La sugerencia tres realizada por el grupo IDETI donde se recomienda determinar el tipo de error, debe ser trabajada en el futuro próximo, debido a su importancia para facilitar el desarrollo de software y que se trata de un tema fuera del alcance de esta investigación, puede ser adecuado aplicar Error-Locating Arrays que permiten determinar y localizar el error.

- Los testers sugirieron que las configuraciones para una prueba pudieran cargarse desde un archivo en Excel en vez de configurarlas solamente mediante la interfaz principal del complemento y que el código generado se copiara automáticamente sobre un proyecto de pruebas.

7 REFERENCIAS BIBLIOGRÁFICAS

[1] A. L. G. a. Hernández. (28 de Febrero de 2013, Un Algoritmo de Optimización Combinatoria para la Construcción de Covering Arrays Mixtos de Fuerza Variable. Available: http://www.tamps.cinvestav.mx/defensa_2013_4

[2] S. Nidhra and 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.

[3] B. P. L. Macario Polo Usaola, Pedro Reales Mateo, *TÉCNICAS COMBINATORIAS Y DE MUTACIÓN PARA TESTING DE SISTEMAS SOFTWARE*, 2012.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software,* vol. 86, pp. 1978-2001, 2013.

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

[6] B. S. Ahmed, M. A. Sahib, and M. Y. Potrus, "Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing," *Engineering Science and Technology, an International Journal*.

[7] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Locating Errors Using ELAs, Covering Arrays, and Adaptive Testing Algorithms," *SIAM Journal on Discrete Mathematics,* vol. 23, pp. 1776-1799, 2010.

[8] Y. Jun and Z. Jian, "Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing," in *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, 2006, pp. 385-394.

[9] D. R. Kuhn and V. Okun, "Pseudo-Exhaustive Testing for Software," in *Software Engineering Workshop, 2006. SEW '06. 30th Annual IEEE/NASA*, 2006, pp. 153-158.

[10] D. R. Kuhn, D. R. Wallace, and J. AM Gallo, "Software fault interactions and implications for software testing," *Software Engineering, IEEE Transactions on,* vol. 30, pp. 418-421, 2004.

[11] B. S. Ahmed and K. Z. Zamli, "A review of covering arrays and their application to software testing," *Journal of Computer Science,* vol. 7, p. 1375, 2011.

[12] S. Maity, "Software Testing with Budget Constraints," in *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, 2012, pp. 258-262.

[13] C. J. Colbourn, "Covering arrays from cyclotomy," *Designs, Codes and Cryptography,* vol. 55, pp. 201-219, 2010.

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

[15] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on,* vol. 32, pp. 20-34, 2006.

[16] M. B. Cohen, C. J. Colbourn, and A. C. Ling, "Augmenting simulated annealing to build interaction test suites," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, 2003, pp. 394-405.

[17] F. Pino, F. García, and M. Piattini, "Priorización de procesos como apoyo a la mejora de procesos en pequeñas organizaciones software," in *XXXIII Conferencia Latinoamericana de Informática (CLEI 2007)*, 2007.

[18] F. J. Pino, F. García, and M. Piattini, "Proceso de Valoración para la Mejora de Procesos Software en Pequeñas Organizaciones," in *CIbSE*, 2008, pp. 211-224.
[19] M. Brcic and D. Kalpic, "Combinatorial testing in software projects," in *MIPRO, 2012 Proceedings of the 35th International Convention*, 2012, pp. 1508-1513.
[20] K. Paternina Palacio and D. Ribón, "Calidad de software:“aplicación en las industrias desarrolladoras de Colombia”," *INGENIATOR,* vol. 2, 2011.
[21] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on,* vol. 32, pp. 20-34, 2006.
[22] C. J. Colbourn, C. Shi, C. Wang, and J. Yan, "Mixed covering arrays of strength three with few factors," *Journal of Statistical Planning and Inference,* vol. 141, pp. 3640-3647, 2011.
[23] A. Rodriguez-Cristerna and J. Torres-Jimenez, "A Simulated Annealing with Variable Neighborhood Search Approach to Construct Mixed Covering Arrays," *Electronic Notes in Discrete Mathematics,* vol. 39, pp. 249-256, 2012.
[24] M. E. V. de Abadia, "Procedimientos para prueba de software," *Publicaciones Icesi,* 2010.
[25] G. T. Daich, G. Price, B. Ragland, and M. Dawood, "Software Test Technologies Report," DTIC Document1994.
[26] B. R., "Managing the Testing Process, 2nd Edition," 2002.
[27] P. A. Barrientos, "Enfoque para pruebas de unidad basado en la generación aleatoria de objetos," Facultad de Informática, 2014.
[28] E. Kit and S. Finzi, *Software testing in the real world: improving the process*: ACM Press/Addison-Wesley Publishing Co., 1995.
[29] B. Pérez, "Proceso de Testing Funcional Independiente," Tesis de Maestría en Informática, Directora: Szasz, N., PEDECIBA Informática, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2006.
[30] J. Ponce, F. Domínguez-Mayo, M. Escalona, M. Mejías, D. Pérez, G. Aragón, and I. Ramos, "Pruebas de aceptación en sistemas navegables," *Innovación, Calidad e Ingeniería del Software,* vol. 6, p. 47, 2010.
[31] M. Cristiá, "Introducción al testing de software," *Universidad nacional de Rosario,* 2009.
[32] J. M. F. Peña. (2006). *Métodos combinatorios*. Available: http://www.uv.mx/personal/jfernandez/files/2010/07/Cap2-Particiones.pdf
[33] R. N. Kacker, D. Richard Kuhn, Y. Lei, and J. F. Lawrence, "Combinatorial testing for software: An adaptation of design of experiments," *Measurement,* vol. 46, pp. 3745-3752, 2013.
[34] S. Choi, H. K. Kim, and D. Y. Oh, "Structures and lower bounds for binary covering arrays," *Discrete Mathematics,* vol. 312, pp. 2958-2968, 2012.
[35] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete applied mathematics,* vol. 138, pp. 143-152, 2004.
[36] D. Navarro, L. G. Cruz, J. G. R. Torres, A. R. Saldivar, S. E. Schaeffer, J. Torres-Jimenez, A. Rodriguez-Cristerna, R. F. Brena, E. García-Ceja, and R. Z. Cabada, "Inteligencia artificial: una reflexión obligada," *Inteligencia artificial: una reflexión obligada,* 2014.
[37] J. Stardom, "Metaheuristics and the search for covering and packing array department of mathematics," *Simon Fraser University, Canada,* vol. 89, 2001.

[38] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software,* vol. 83, pp. 60-66, 2010.

[39] L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing," *Software Engineering, IEEE Transactions on,* vol. SE-6, pp. 247-257, 1980.

[40] E. Díaz, J. Tuya, R. Blanco, and J. Javier Dolado, "A tabu search algorithm for structural software testing," *Computers & Operations Research,* vol. 35, pp. 3052-3072, 2008.

[41] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems,* vol. 29, pp. 1758-1773, 2013.

[42] A. Hessel and P. Pettersson, "A Global Algorithm for Model-Based Test Suite Generation," *Electronic Notes in Theoretical Computer Science,* vol. 190, pp. 47-59, 2007.

[43] C. Cheng, A. Dumitrescu, and P. Schroeder, "Generating small combinatorial test suites to cover input-output relationships," in *Quality Software, 2003. Proceedings. Third International Conference on*, 2003, pp. 76-82.

[44] J. YAN and J. ZHANG, "Combinatorial testing: principles and methods," *Journal of Software,* vol. 20, pp. 1393-1405, 2009.

[45] A. Baartmans and S. Sane, "Blocker sets, orthogonal arrays and their application to combination locks," *Discrete Mathematics,* vol. 308, pp. 2885-2895, 2008.

[46] K. Vengadesan, T. Anbupalam, and N. Gautham, "An application of experimental design using mutually orthogonal Latin squares in conformational studies of peptides," *Biochemical and Biophysical Research Communications,* vol. 316, pp. 731-737, 2004.

[47] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for t-way software testing," in *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*, 2007, pp. 549-556.

[48] A. Calvagna and A. Gargantini, "IPO-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, 2009, pp. 10-18.

[49] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability,* vol. 18, pp. 125-148, 2008.

[50] M. F. Klaib, S. Muthuraman, N. Ahmad, and R. Sidek, "Tree based test case generation and cost calculation strategy for uniform parametric pairwise testing," *Journal of Computer Science,* vol. 6, p. 542, 2010.

[51] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *Software Engineering, IEEE Transactions on,* vol. 23, pp. 437-444, 1997.

[52] M. B. Cohen, "Designing test suites for software interaction testing," Citeseer, 2004.

[53] J. Czerwonka, "Pairwise Testing in Real World Practical Extensions to Test Case Generators."

[54] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Software Testing, Verification and Reliability,* vol. 17, pp. 159-182, 2007.

[55] E. Lehmann and J. Wegener, "Test case design by means of the CTE XL," in *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark*, 2000.

[56] Y. Yu, S. P. Ng, and E. Y. Chan, "Generating, selecting and prioritizing test cases from specifications with tool support," in *Quality Software, 2003. Proceedings. Third International Conference on*, 2003, pp. 83-90.

[57] Y.-W. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *Aerospace Conference Proceedings, 2000 IEEE*, 2000, pp. 431-437.

[58] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Information and Software Technology,* vol. 51, pp. 957-976, 2009.

[59] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, 2004, pp. 72-77.

[60] B. S. Ahmed and K. Z. Zamli, "PSTG: a t-way strategy adopting particle swarm optimization," in *Mathematical/Analytical Modelling and Computer Simulation (AMS), 2010 Fourth Asia International Conference on*, 2010, pp. 1-5.

[61] P. Nayeri, C. J. Colbourn, and G. Konjevod, "Randomized post-optimization of covering arrays," *European Journal of Combinatorics,* vol. 34, pp. 91-103, 2013.

[62] N. Tracey, J. A. Clark, K. Mander, N. Tracey, J. A. Clark, and K. Mander, "Automated Program Flaw Finding Using Simulated Annealing," 1998, pp. 73-81.

[63] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on Tabu search," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 2003, pp. 310-313.

[64] I. I. Márquez. (Agosto 2013). *Construcción de Torres de Covering Arrays*. Available: http://www.tamps.cinvestav.mx/defensa_2013_7

[65] P. Danziger, E. Mendelsohn, L. Moura, and B. Stevens, "Covering arrays avoiding forbidden edges," *Theoretical Computer Science,* vol. 410, pp. 5403-5414, 2009.

Anexo A: Artículo "Complemento De Vs.Net Para La Definición De Pruebas De Software De Caja Negra Mediante Arreglos De Cobertura"

Anexo B: Documento Con Las Sugerencias Otorgadas Por El Grupo IDETI

Anexo C (Digital): Encuestas Con Sus Respectivas Tabulaciones Y Gráficas, Código Fuente, Script SQL-Server, Instaladores Del Complemento Para VS.Net 2010, 2012, 2013 y Video Tutoriales