Modelo para Pruebas Automáticas de Software aplicando un Agente de Evolución Flexible

C. Jordán[1], A. Zúñiga[2], D. Medina[3]
Facultad de Ingeniería en Electricidad y Computación
Escuela Superior Politécnica del Litoral
Campus Gustavo Galindo, Km. 30.5 Vía Perimetral
Apartado 09-01-5863, Guayaquil-Ecuador
cjordan@fiec.espol.edu.ec[1], azuniga@espol.edu.ec[2], mrdumax@gmail.com[3]

Resumen

El objetivo de este proyecto es la aplicación de un nuevo modelo de Computación Evolutiva conocido como Agente de Evolución Flexible AEF, para mejorar el proceso de las pruebas de software; la etapa de pruebas en el proceso de desarrollo de proyectos de software es muy costosa en términos económicos y de esfuerzo. Este trabajo se enfoca en las pruebas estructurales de software con el fin de reducir el esfuerzo y tiempo que demandan las pruebas desarrolladas generalmente de forma manual o en forma aleatoria. El modelo propuesto representa el problema del testing como un problema de optimización, definiendo una función objetivo para determinar los casos de prueba óptimos que ejecutan una condición determinada del programa. El problema de optimización es resuelto mediante la aplicación del AEF, que ha presentado mejores resultados que los algoritmos evolutivos tradicionales en varias aplicaciones en optimización. Los resultados obtenidos serán comparados con otras técnicas evolutivas empleadas para las pruebas; el modelo desarrollado ha sido aplicado a un programa de prueba comúnmente empleados en trabajos similares.

Palabras Claves: *Pruebas automáticas de software, algoritmos evolutivos, agente de evolución flexible.*

Abstract

The aim of this project is the application of a novel metaheuristic technique called Flexible Evolutionary Agent (AEF), to improve the process of automatic test data generation for software testing; testing stage in the software development process is very expensive in terms of economy and human effort. This project is focused in the structural software testing, in order to reduce the effort and time taken by tests, generally developed in manual form and applying random methods. The proposed model represents the testing problem like an optimization problem, defining an objective function to determine the optimal test cases which executes a program's condition. The optimization problem is solved by application of AEF, which has been presented better results than other evolutionary techniques in several optimization problems. The results obtained will be compared with others from classical and evolutionary models of data test generation; the model is applied on a selected program, which has been commonly used in similar works.

Keywords: *Automatic software testing, evolutionary algorithms, evolutionary flexible agent.*

Recibido: Junio, 2008
Aceptado: Agosto, 2008

1. Introducción

La etapa de pruebas es una actividad muy importante en el proceso de desarrollo de software. El objetivo de las pruebas es descubrir errores y asegurar que el comportamiento del programa es correcto de acuerdo a criterios específicos de las pruebas [10]. Alrededor del 40% de los costos totales del desarrollo del software se consumen en dicha etapa; esta etapa es muy costosa no solo económicamente, sino también en recursos humanos, tiempo e intensidad de trabajo, muchas veces sin ninguna contribución en términos de funcionalidad. Idealmente, las pruebas del software deben garantizar la ausencia de fallas, pero en realidad sólo revelan su existencia sin ninguna garantía de su ausencia total.

Los modelos de pruebas de software pueden clasificarse en modelos de pruebas funcionales y pruebas estructurales.

Las pruebas funcionales también son conocidas como "pruebas basadas en especificaciones" o simplemente "pruebas de caja negra"; el software es considerado como una caja negra y su funcionalidad es probada proporcionando varias combinaciones de datos de pruebas [11], sin conocimiento acerca de la estructura interna del programa.

La fortaleza de las pruebas de caja negra es que las pruebas pueden derivarse de forma temprana durante el desarrollo; esto permite detectar fallas lógicas no consideradas.

Las pruebas estructurales son también conocidas como "pruebas de caja blanca"; corresponden al proceso de derivar pruebas a partir de la estructura interna del programa objeto. Si la cantidad de trayectorias en el diagrama de flujo es muy grande, no es posible probar sistemáticamente todas las posibles trayectorias (un programa de 100 líneas de código en C con dos lazos anidados puede contener hasta 1014 trayectorias diferentes).

Muchas formas de pruebas estructurales hacen referencia al "diagrama de flujo" del programa (DF). Un diagrama de flujo DF es un gafo dirigido $G(N,E,s,e)$, donde N es el conjunto de nodos, E es el conjunto de arcos, y s y e son los nodos únicos de entrada y salida del grafo. Cada nodo $n \in N$ es una condición en el programa, donde cada arco $e(n_i,n_j) \in E$, representa la transferencia del control desde el nodo n_i hacia el nodo n_j.

Comúnmente las pruebas de software son desarrolladas mediante el ingreso automático de datos aleatorios o simplemente son llevadas a cabo por humanos, lo que dificulta la cobertura de una cantidad importante de condiciones que permitan descubrir errores.

En las siguientes secciones se presentan los criterios de las pruebas automáticas (sección 2), una descripción del modelo de pruebas empleado (sección 3), una descripción del Agente de Evolución Flexible (Sección 4) y los resultados de las pruebas preliminares realizadas en un programa objeto empleados en la literatura de pruebas de software.

2. Pruebas automáticas de software

Inicialmente los modelos de automatización de pruebas se encaminaron a la determinación de los datos de pruebas óptimos para un software determinado. En las dos últimas décadas se ha desarrollado una importante cantidad de trabajos en esta área; en los 80's, la mayoría de los métodos desarrollados eran manuales y empleaban la evaluación simbólica de ecuaciones asociadas a los condiciones del código para determinar los datos de prueba [9]; estos métodos tenían la dificultad de resolver las ecuaciones propuestas mediante manipulaciones algebraicas complicadas.

La aparición de métodos de optimización basados en técnicas de búsqueda ofreció una mejoría al proceso de pruebas estructurales.

Uno de los primeros intentos por combinar las pruebas estructurales con técnicas de búsqueda fue propuesto por Milles & Spooner [8]; los autores seleccionan una trayectoria del DF y las sentencias de la rama se reemplazan por una "restricción funcional" de la trayectoria; una función f proporciona una medida real de la cercanía de las soluciones calculadas del valor óptimo que satisface dichas restricciones [8].

Bogdan Korel [7], extendió las ideas presentadas por Milles & Spooner y, en 1990, propuso la aplicación de métodos de minimización de funciones basados en búsqueda directa secuencial para determinar los datos óptimos de prueba; su propuesta se basa en la definición de una función de variable real asociada con cada condición del programa y cuya estructura depende del tipo de operador relacional de la condición [7]; esta función, que depende de las variables de entrada del programa, toma valores positivos cuando la condición asociada es falsa y negativos cuando la condición asociada es verdadera.

Así, entonces, se considera que este trabajo de Korel es el punto de partida para el desarrollo de los nuevos modelos para realizar pruebas automáticas de software, pues plantea dicho problema como uno de optimización, que puede ser resuelto por una gran cantidad de métodos conocidos; por ejemplo, los métodos clásicos de optimización basados en programación lineal y no lineal o los métodos de búsqueda directa; sin embargo, la aplicación de estas técnicas requiere el uso de funciones continuas y generalmente se requiere información de sus derivadas, y por tanto son susceptibles de quedar atrapadas en óptimos locales. El éxito de las técnicas

metaheurísticas como métodos de optimización global, especialmente en problemas con funciones multimodales y no diferenciables, han llamado la atención de los investigadores en el área de pruebas de software [5,6].

3. Modelo de pruebas empleado

El modelo de pruebas empleado en este trabajo se basa en el método propuesto por el grupo de investigaciones en Metaheurísticas de la Universidad de Málaga, dirigido por Enrique Alba [1,2] y presentado por Francisco Chicano en su Tesis Doctoral [4]; la propuesta de Chicano emplea poblaciones de casos de pruebas y realiza una búsqueda de los mejores casos mediante algoritmos evolutivos.

El modelo propuesto en este trabajo, está conformado por dos componentes principales: el programa objeto instrumentado y el módulo generador.

El programa objeto se modifica adicionando líneas de código en cada condición, lo que permite conocer el status de cada condición al presentarse un caso de prueba (si fue evaluada con verdadero o falso o no fue evaluada) y el valor de la función de fitness asociado a dicha condición.

El valor de la función fitness que retorna el programa objeto para un caso de prueba dado se calcula utilizando dos funciones: una función $distancia(x)$ que mide la distancia entre los argumentos de la condición (denominada objetivo parcial), cuyos valores corresponden a los datos de prueba que ejecutan dicha condición en uno de sus sentidos (verdadero o falso). Las funciones de distancia se presentan en la tabla 1.

Tabla 1. Función de distancia para varias estructuras de condiciones [4]

Condición	$distancia(x)$ condición verdadera	$distancia(x)$ condición falso
$a < b$	$a - b + 1$	$b - a$
$a \leq b$	$a - b$	$b - a + 1$
$a = b$	$(b-a)^2$	$\left(1+(b-a)^2\right)^{-1}$
$a \neq b$	$\left(1+(b-a)^2\right)^{-1}$	$(b-a)^2$
a	$\left(1+a^2\right)^{-1}$	a^2

La segunda función definida por Chicano es la función de fitness, que será empleada en el método de optimización para evaluar las soluciones [4], donde la función de distancia depende de la condición evaluada (tabla 1):

$$fitness(x) = \pi/2 - \arctan\left(distancia(x)\right) + 0.1 \quad \textbf{(1)}$$

El módulo generador se encarga de realizar una búsqueda del mejor caso de prueba que ejecuta cada condición del programa objeto; cada condición se considera como un objetivo parcial del problema, por lo que su función de fitness asociada se optimiza empleando AE's; una vez evaluada una condición se almacena el resultado de la evaluación que servirá luego para determinar el criterio de cobertura de las pruebas del programa objeto.

El criterio considerado para las pruebas está representado por la cobertura de condiciones, expresada en porcentaje del total de condiciones del programa objeto, mediante la ecuación 2 [1]:

$$\%Cov = \left(1 - \frac{\#\ condiciones\ cubiertas}{\#\ total\ condiciones}\right) * 100\% \quad \textbf{(2)}$$

Se define la cobertura de una condición cuando ha sido ejecutada en sentido verdadero y falso durante el proceso de pruebas; si una condición ha sido ejecutada en un sentido se dice que la condición ha sido alcanzada y si no ha sido ejecutada, se dice que la condición no ha sido alcanzada.

Si una condición no ha sido alcanzada, se continúa con la siguiente condición de acuerdo a su ubicación en el diagrama de flujo del programa objeto; puede darse el caso que durante la optimización de una condición particular, uno de los casos de prueba de la población alcance o cubra otra condición.

Durante el proceso se define una tabla de cobertura de condiciones, la cual se actualiza después de la evaluación de cada objetivo parcial; al final de la ejecución de las pruebas, esta tabla proporciona el porcentaje de cobertura de los casos de prueba generados por el modelo.

4. Agente de Evolución Flexible

Los Algoritmos Evolutivos AE's son técnicas de búsqueda estocástica basadas en la genética natural y los mecanismos de la evolución. Uno de los aspectos más interesantes de los AE's es que no requieren ningún conocimiento previo sobre el problema evaluado, limitaciones del espacio de soluciones o propiedades especiales como convexidad, unimodalidad o existencia de derivadas, lo que permiten su aplicación en una amplia variedad de problemas.

Los AE's trabajan sobre conjuntos de soluciones (poblaciones), emulando los procesos naturales de la evolución mediante operadores de selección (se consideran las mejores soluciones), operadores de

cruce (se combinan estas mejores soluciones para obtener nuevas soluciones), operadores de mutación (se modifican las nuevas soluciones para evitar replicaciones) y por último la supervivencia del más apto (competición entre soluciones padres e hijos). Cada uno de los operadores de un AE contiene parámetros asociados que deben ser ajustados de acuerdo a cada problema abordado.

Este ajuste de parámetros, conocido como "adaptación", puede ser realizado manual o automáticamente; el ajuste automático de parámetros se conoce como auto-adaptación, y es uno de los tópicos de mayor estudio dentro de los AE's, debido a que problemas de una misma familia requieren operadores diferentes y ajuste de parámetros diferentes.

Existe una gran cantidad de propuestas para la autoadaptación de parámetros, entre las que se destacan los métodos de auto-adaptación de las Estrategias Evolutivas y la propuesta de los Agentes de Evolución Flexible AEF; mecanismos de auto-adaptación más eficientes permiten mejorar el rendimiento de los algoritmos evolutivos [12].

Estudios realizados con varios problemas de optimización clásicos demuestran la superioridad de estos dos paradigmas de la computación evolutiva sobre otros como los algoritmos genéticos.

El objetivo de un AEF es aprovechar las nuevas tendencias de los algoritmos evolutivos en lo referente a mecanismos de auto-adaptación, además de aprovechar otras líneas de investigación en el área de toma de decisiones.

La implementación se ha dividido en "motores" que determinan las tareas a ser ejecutadas, y que a su vez, se encuentran gobernados por un Motor de Decisión MCC. En la figura 1 se muestra la estructura general de un AEF y que ha sido tomada de [3].

El algoritmo gira en torno al motor principal que es el Motor de Decisión MCC, que se encarga de dirigir todas las acciones que se llevarán a cabo durante el proceso de optimización. Cada uno de los motores retroalimentan al MCC durante el proceso, permitiendo que el MCC tome las decisiones adecuadas respecto a los mecanismos de selección, cruce, mutación, aprendizaje, filtrado y otros mecanismos que pueden ser definidos, en cada generación para obtener nuevas y mejores muestras respecto a la generación anterior.

El algoritmo inicia con la generación de una población inicial generada aleatoriamente, luego de lo cual se realiza la evaluación de la función fitness. Luego el algoritmo ejecuta el motor de aprendizaje que almacena toda la información útil sobre el estado de la optimización y realiza cálculos de medidas estadísticas que permitirán conocer ciertas características de la evolución de las soluciones durante el desarrollo de la optimización.

Una vez determinada la información del motor de aprendizaje, se ejecuta el motor de selección, donde se determinan las soluciones que van a ser modificadas mediante los operadores genéticos considerados en la implementación; en este motor se seleccionan individuos y variables que serán empleadas para generar nuevas soluciones y finalmente determina con qué operador de muestreo se obtendrán estas nuevas soluciones.

En esta implementación no se emplean parámetros propios de un algoritmo evolutivo como la tasa de cruce o mutación, ya que el MCC junto con el motor de selección, determinarán la aplicación de un operador sin necesidad de ningún parámetro adicional.

Todas las decisiones tomadas en le motor de selección son comandadas por el MCC, donde se define una regla probabilística de selección tanto de individuos como de operadores genéticos.

El motor de muestreo es el encargado de realizar las operaciones genéticas entre los individuos seleccionados empleando los operadores determinados por el motor de selección. Los motores de decisión, selección y muestreo constituyen el corazón del agente de evolución flexible pues es en esta etapa que se determinan las nuevas soluciones y los operadores genéticos empleados para generarlas.

En la teoría de los AEF, se define también un motor de filtrado que sirve para corregir cualquier error que se detecte en las soluciones generadas, eliminando, por ejemplo, soluciones repetidas o individuos infactibles.

El algoritmo termina al cumplirse un criterio de parada como el alcanzar el valor óptimo o alcanzar un número máximo de generaciones, o cualquiera de los criterios de parada empleados comúnmente en los algoritmos evolutivos clásicos.

Cada uno de los mecanismos expuestos posee características especiales que deben ser explicadas individualmente.

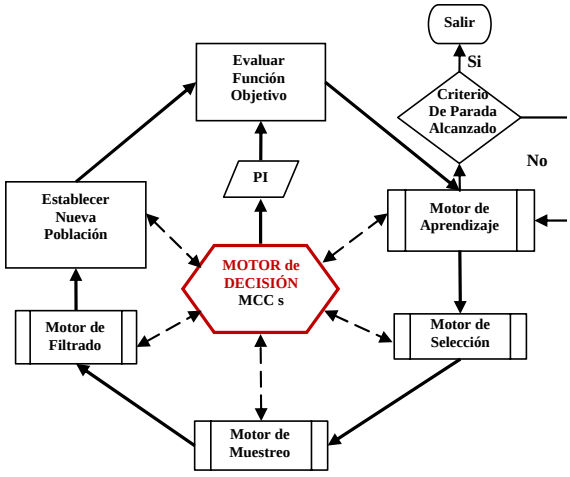

Figura 1. Esquema de un Agente de Evolución Flexible.

El esquema presentado en la figura 1 constituye el esquema general de un AEF, habiéndose presentado varias propuestas y actualizaciones del AEF por parte de los investigadores del CEANI [3,12].

5. Algoritmos de prueba

Para realzar las pruebas del modelo propuesto, se ha seleccionado un algoritmo empleado por el grupo de Alba y Chicano [1, 2 y 3] en sus trabajos; el programa objeto, denominado "Triangulo", clasifica triángulos dados sus tres lados; este algoritmo tiene 10 condiciones y tres variables de entrada.

Este programa tomado de [4] fue modificado ligeramente por medio de introducir manualmente unas pocas instrucciones, con el fin de informar al módulo generador de casos de prueba acerca del cumplimiento de los objetivos parciales, y el valor de la función fitness para cada caso probado.

6. Resultados experimentales

Se realizó la implementación de un prototipo para pruebas, que consta de un módulo principal que gestiona tanto el programa objeto como el módulo de optimización.

El módulo de optimización contiene 4 algoritmos, dos algoritmo genéticos con codificación real (GA-25/50 con operador de cruce de un punto y GA-BLX-α con operador de cruce blend crossover), un algoritmo de estrategias evolutivas con autoadaptación local sin rotación de ejes (ES 25/100) y el agente de evolución flexible AEF(25,50).

Se realizaron 20 ejecuciones del modelo de pruebas, considerando una población de 25 individuos y un máximo de 50 generaciones para los algoritmos evolutivos.

La tabla 2 muestra que los mejores resultados se consiguen con los modelos de pruebas empleando las estrategias evolutivas (cobertura 96.75%) y el agente de evolución flexible (cobertura 99.50%); el algoritmo genético con operador de cruce BLX-α (94.25%) presenta mejor desempeño que el algoritmo genético con cruce de un punto (83.50%). La figura 2 muestra gráficamente los resultados obtenidos.

Tabla 2. Resultado de las simulaciones para el programa clasificador de triángulos

Método	Número de casos probados	Porcentaje de cobertura
GA(25,50)	6806.40	83.50%
GA-BLX-α	2523.85	94.25%
ES(25/100)	1585.25	96.75%
AEF(25,50)	1753.00	99.50%

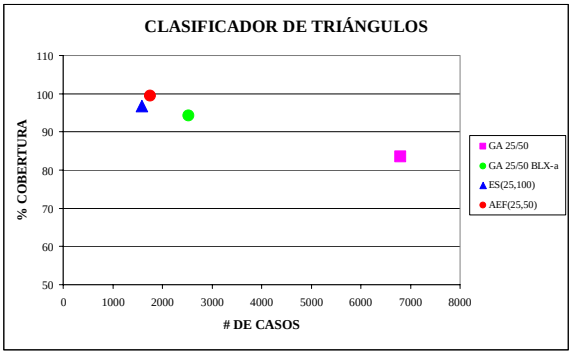

Figura 2. Resultado de las pruebas en programa clasificador de triángulos.

Un análisis detallado de las simulaciones para el GA-BLX-α, las EE's y el AEF muestran un comportamiento muy homogéneo, con coberturas superiores al 90%; la distribución de las simulaciones con el GA-25/50 presenta una dispersión más acentuada con porcentajes de cobertura menores al 90% en la mayoría de los casos (el 80% de las simulaciones).

7. Conclusiones

El objetivo de este trabajo fue la aplicación de un Agente de Evolución Flexible para resolver el problema de pruebas de software; el AEF ha demostrado superioridad frente a otros métodos evolutivos en problemas de optimización.

El modelo fue probado en un programa objeto usado frecuentemente en la literatura de pruebas automáticas. Los resultados demuestran una superioridad del AEF frente a las otras técnicas evolutivas como el algoritmo genético y las estrategias evolutivas.

Un análisis de las simulaciones individuales, muestra una menor dispersión de las coberturas para 20 ejecuciones empleando el AEF, lo que demuestra la homogeneidad de las soluciones obtenidas.

La aplicación del modelo de pruebas basado en el AEF puede mejorar sustancialmente los procesos de prueba en programas de gran tamaño.

8. Agradecimientos

Este trabajo cuenta con el soporte financiero del Proyecto VLIR-ESPOL, Componente 8.

Los autores agradecen el soporte técnico de los Profesores: Dra. Monique Snöeck de la Universidad Católica de Leuven y Dr. Blas Galván de la Universidad de Las Palmas de Gran Canaria.

9. Referencias

[1] Alba, E., Chicano, F., “Software testing using evolutionary strategies”, The 2nd Workshop on Rapid Integration of Software Engineering Techniques (RISE 05), LNCS3943, Greece, September 2005, pp.56-65.

[2] Alba, E., Chicano, F., Janson, S., “Testeo de software con dos técnicas metaheurísticas”, in VX Jornadas de Ingeniería de Software y Bases de Datos JISBD 2006, José Riquelme-Pere Botella eds., Barcelona, 2006.

[3] Alonso, S., “Propuesta de un método evolutivo flexible de optimización global”, Tesis Doctoral, Universidad de Las Palmas de Gran Canaria, España, 2006.

[4] Chicano, J. F., “Metaheurísticas e Ingeniería del Software”, Tesis Doctoral, Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, España, 2007.

[5] Clarke, J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Rees, K., Roper, M., Shepperd, M., “The Application of Metaheuristic Search Techniques to Problems in Software Engineering”, Technical Report SEMINAL-TR-01-2000, Brunel University, United Kingdom, August 2000.

[6] Harman, M., “Search Based Software Engineering”, Information and Software Technology, Vol. 43, No. 14, December 2001, pp. 833-839.

[7] Korel, B., “Automated Software Test Data Generation”, IEEE Transactions on Software Engineering, Vol. 16, No. 8, August 1990, pp. 870-879.

[8] Miller, W., Spooner, D., “Automatic generation of floating-point test data”, *IEEE Transactions on Software Engineering* 2, no. 3, 1976, pp. 223 – 226.

[9] Pei, M., Goodman, E. D., Zongyi, G., Zhong, K., “Automated Software Test Data Generation Using A Genetic Algorithm”, Technical Report GARAGE, Michigan State University, June 2004.

[10] Pressman, R. S., Software engineering: a practitioner’s approach. McGraw-Hill, fifth edition, 2001. ISBN 0-07-365578-3.