Generación de Secuencias de Pruebas Funcionales con Algoritmos Bio-inspirados

Javier Ferrer, Peter M. Kruse, Francisco Chicano y Enrique Alba

Resumen— **La generación de secuencias de pruebas dinámicas desde una especificación formal complementa los métodos tradicionales de pruebas para encontrar errores en el código fuente. En este artículo extendemos un enfoque combinatorio en particular, el llamado Método del Árbol de Clasificación -** *Classification Tree Method* **(CTM), con información sobre transiciones para la generación de secuencias de pruebas. Aunque en este artículo usamos CTM, esta extensión es también posible para otros métodos de pruebas combinatorios. La generación de secuencias mínimas de pruebas que cumplan con el criterio de cobertura requerido es un problema NP-duro. Así que, vamos a necesitar un enfoque basado en búsqueda para encontrar las secuencias de pruebas óptimas en un tiempo razonable. En la sección de experimentos se comparan dos técnicas bio-inspiradas con un algoritmo voraz determinista. Hemos utilizado un conjunto de 12 modelos de programas jerárquicos concurrentes, extraídos de la literatura. Nuestras propuestas basadas en algoritmos genéticos y colonias de hormigas son capaces de generar secuencias de pruebas que alcanzan cobertura total tanto de clases como de transiciones, es decir, son capaces de encontrar el camino válido más corto para visitar todas las clases y usar todas las transiciones de un programa. El análisis de los experimentos revela que las propuestas bio-inspiradas son mejores que el algoritmo voraz determinista, especialmente en las instancias más complejas.**

Palabras clave— **Colonias de Hormigas, Algoritmos Genéticos, Generación de Secuencias de Pruebas, Pruebas Funcionales, Ingeniería del Software basada en Búsqueda.**

I. Introducción

La fase de pruebas es una etapa muy importante en el ciclo de vida del software, cuyo objetivo es asegurar cierta calidad en el software. En [1] se detalla el alto impacto de una fase de pruebas inadecuada. Además, se estima que la mitad del tiempo de desarrollo y más de la mitad del coste total de un proyecto se dedica a las pruebas del software [2]. La automatización de la generación de pruebas podría reducir el coste del proyecto completo, esto explicaría el interés a nivel industrial y académico en dicha automatización. La generación de unas pruebas adecuadas implica un gran esfuerzo computacional, de modo que necesitamos enfoques basados en búsqueda para enfrentarnos a este problema. Actualmente, la generación automática de casos de prueba es uno de los temas más estudiados en el dominio conocido como *Search-Based Software Engineering* (SBSE) [3].

Los algoritmos más utilizados para la generación automática de casos de prueba son los algoritmos evolutivos (EAs) [4], de hecho, se ha acuñado el término *evolutionary testing* para referirse a este enfoque. El paradigma de pruebas estructurales ha recibido mucha atención usando EAs, pero el uso de técnicas de búsqueda en pruebas funcionales es menos frecuente [5]. Esto se debe a la naturaleza de la especificación, que se suele escribir en lenguaje natural.

Tradicionalmente, el desafío ha sido generar conjuntos de pruebas para verificar completamente el software. No es posible realizar pruebas exhaustivas en grandes proyectos [6], así que hay que seleccionar un subconjunto de todas las posibles pruebas. Las pruebas de interacción combinatoria (CIT) [7] intentan atajar este problema. El enfoque CIT intenta encontrar el conjunto mínimo de pruebas que consiga la cobertura deseada. En general, esta tarea consiste en la generación de todas las posibles combinaciones de los valores de los parámetros (este tarea es NP-dura [8]). La fortaleza de este enfoque, *t-strength*, depende del número (t) de parámetros involucrados en las combinaciones (p. ej. $t = 2$ para pares, $t = 3$ para trios, etc...). Aunque este tipo de pruebas se han estudiado ampliamente, existen dos cuestiones relacionadas que no han sido abordadas en la literatura: las dependencias entre casos de prueba individuales y el estado del software que se está probando (*Software Under Test* - SUT).

En multitud de ocasiones se requiere que el software se encuentre en un estado concreto para probar una funcionalidad. De hecho, en sistemas software grandes, el coste derivado de situar al programa en cierto estado puede ser importante. Por ejemplo, probar el sistema ABS de un vehículo requiere que éste alcance cierta velocidad. En este caso, tiene sentido considerar la generación de secuencias de pruebas que nos permitan probar una funcionalidad (aceleración del vehículo) mientras que modificamos el estado del SUT (considerando las dependencias entre casos de prueba). El ahorro implícito de usar esta técnica es la razón por la cual la generación de secuencias es relevante y merece más esfuerzo de investigación.

En este trabajo presentamos dos enfoques bio-inspirados para generar secuencias de pruebas automáticamente. Estas técnicas son capaces de encontrar soluciones cercanas al óptimo usando una cantidad de recursos razonables. Hemos comparado el comportamiento de dos algoritmos bio-inspirados con respecto a un algoritmo voraz determinista exis-

Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain. email:{ferrer,chicano,eat} at lcc.uma.es

Berner & Mattner Systemtechnik GmbH, Berlin, Germany. email: peter.kruse at berner-mattner.com

tente [9]. La primera de ellas es un algoritmo genético llamado Generador Genético de Secuencias de Pruebas (GGSP). La segunda es una colonia de hormigas, especialmente diseñada para tratar con grandes grafos que llamaremos Generador de Secuencias con Colonias de Hormigas (GSCH). A continuación vamos a detallar las contribuciones principales de este trabajo:

- Extendemos el Método del Árbol de Clasificación [10] con información de las transiciones para poder generar las secuencias de pruebas más cortas sobre modelos de programas, a este enfoque lo llamamos *Extended Classification Tree Method* (ECTM).
- Comparamos el rendimiento de tres enfoques diferentes (algoritmo genético, colonia de hormigas y algoritmo voraz determinista) en el problema de generación de secuencias de pruebas.
- Hemos experimentado con un total de 12 programas, 2 tipos de coberturas (la cobertura de clases y la cobertura de transiciones) y 3 técnicas de resolución.

Este artículo se organiza de la siguiente manera. En la sección II presentamos la técnica CTM: cómo esta diseñada, cómo la extendemos y qué criterio de adecuación usamos. La sección III describe el problema de generación de secuencias de pruebas. En la sección IV presentamos las propuestas bio-inspiradas y resumimos el algoritmo voraz determinista re-implementado para poder realizar la comparación. La sección V está dedicada a la presentación del conjunto de programas a probar y al análisis de los resultados de los tres enfoques. Por último, la sección VI presenta algunas conclusiones y trabajo futuro.

II. Método del Árbol de Clasificación

El CTM [10] se puede usar para la identificación de casos de prueba para pruebas funcionales sobre todos los niveles (p. ej. pruebas a nivel de componentes, pruebas unitarias, pruebas de integración). Está basado en el método de partición de categorías [11], el cual divide el domino de pruebas en clases disjuntas que representan aspectos importantes del SUT. Estas clases pueden identificarse como los estados del SUT. Aplicar el CTM requiere dos pasos: diseñar el árbol de clasificación y definir los casos de prueba.

A. Diseño del Árbol de Clasificación

El árbol de clasificación está basado en una especificación funcional del SUT. Para cada aspecto de interés (llamado clasificación), el dominio de entrada está dividido en conjuntos disjuntos llamados clases. La Figura 1 ilustra el árbol de clasificación con un ejemplo simple de videojuego. Identificamos dos aspectos de interés (*Game* and *Pause*). Las clasificaciones son refinadas en clases que representan la partición concreta de los valores de entrada. Estas particiones pueden ser refinadas nuevamente en otras clasificaciones y clases de más bajo nivel. En nuestro ejemplo, la clasificación *Playing* es identificada para la clase *runningGame*, y es a su vez subdividida en las clases *startup* y *controlling*.

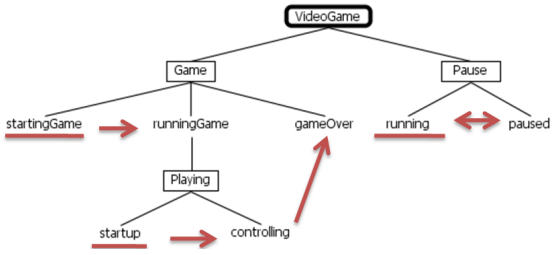

Fig. 1: Un modelo extendido ECTM de un videojuego básico.

Dado un árbol de clasificación, los casos de prueba se definen como combinación de clases de diferentes clasificaciones. Las clasificaciones sólo pueden contener clases disjuntas, así que los casos de prueba no pueden contener varias clases de una misma clasificación. Un caso de prueba para este ejemplo es:

Game:runningGame(*Playing*:startup),
Pause:running

en el cual la clase *running* se selecciona de la clasificación *Pause* y *runningGame* se selecciona de *Game*. Como la clase *runningGame* tiene una clasificación más interna, *Playing*, tenemos que seleccionar una clase de ella, en este caso *startup*.

Una secuencia de pruebas es una lista ordenada de casos de prueba que podría ser visitada de forma secuencial con el objetivo de probar la funcionalidad del sistema completo.

B. Extensiones del Árbol de Clasificación

El árbol de clasificación definido en la sección anterior puede ser usado para diseñar casos de prueba aislados. Sin embargo, el SUT puede tener operaciones relativa a transiciones entre clases del árbol de clasificación, que deben ser ejecutadas para pasar de un estado a otro. Estas operaciones pueden ser modeladas gracias a las transiciones entre clases (ver la Fig. 1). En el mundo real, estas transiciones vienen dadas por la semántica del SUT. También asumimos que cada clasificación tiene una clase por defecto, que destacamos en el gráfico subrayando la clase. Esta extensión del árbol de clasificación (ECTM) se puede ver como una máquina de estados concurrente jerárquica (HCSM) donde las clases equivalen a los estados, y las clasificaciones equivalen a las regiones ortogonales.

C. Criterios de Cobertura

En este artículo hemos considerado dos criterios de cobertura: cobertura de clases y de transiciones. La cobertura de clases consiste en cubrir todas las

clases del árbol de clasificación con los casos de prueba generados. La cobertura de transiciones requiere cubrir todas las transiciones disponibles entre las clases del ECTM. En nuestro ejemplo de la Figura 1, tenemos que cubrir ocho clases en total (*VideoGame, startingGame, runningGame, startup, controlling, gameOver, running, and paused*), y cinco transiciones (*startingGame* → *runningGame*, *startup* → *controlling*, *controlling* → *gameOver*, *running* → *paused*, *paused* → *running*).

III. El Problema de la Generación de Secuencia de Pruebas

El problema de la generación de secuencias de pruebas (TSGP - *Test Sequence Generation Problem*) ha recibido poca atención en la literatura existente, mucha menos que la generación de casos de prueba estructurales. En este artículo hemos extendido CTM, y además de tener en cuenta las restricciones impuestas por la jerarquía clasificación-clase, tenemos en cuenta reglas de dependencia entre los casos de prueba. Estas principalmente vienen impuestas por las transiciones entre clases, y no deben ser violadas por ningún caso de prueba de la secuencia. Lidiar con las reglas de dependencia es importante ya que podemos combinar las pruebas de varios estados, resultando secuencias más cortas. De esta forma necesitaremos menos recursos para probar toda la funcionalidad del SUT.

Nuestro enfoque se basa en la idea propuesta por Conrad [12], quien sugiere la interpretación del árbol de clasificación como una máquina de estados finita paralela. Sin embargo, necesitamos extender el enfoque de Conrad para interpretar las clases refinadas, este concepto es similar a los refinamientos de los estados en UML. Nuestro enfoque se puede asemejar a una máquina de estados, ya que tiene estados concurrentes y existe jerarquía en el modelo. A continuación pasamos a describir el problema de la generación de secuencias de pruebas.

Un caso de prueba para un ECTM es un conjunto de clases que cumplen algunas reglas. En particular, no es posible que existan dos clases que pertenezcan a la misma clasificación. Si una clase refinada pertenece al caso de prueba, entonces debe haber una clase por cada clasificación cuyo padre ha sido refinado. Además, si una clase está presente en un caso de prueba, todas las clases ascendientes en el ECTM deben ser incluidas también. Por ejemplo, el conjunto $Q = \{startingGame, running\}$ es un caso de prueba, pero el conjunto $Q = \{runningGame\}$ no es un caso de prueba porque no hay ninguna clase de las clasificaciones *Pause* y *Playing*.

Podemos transitar de una clase a otra tomando alguna de las transiciones disponibles. El nuevo caso de prueba excluye la clase origen e incluye la clase destino de la transición. Para cumplir con las normas descritas anteriormente, algunas clases del caso de prueba origen podrían desaparecer también y algunas clases adicionales deberán incluirse en el nuevo caso de prueba. Por ejemplo, si tomamos la transición *startingGame* → *runningGame* desde el caso de prueba $Q_1 = \{startingGame, running\}$ en nuestro ejemplo de videojuego, alcanzamos el caso de prueba $Q_2 = \{runningGame, startup, running\}$. Observamos que la clase *startingGame* ha sido eliminada de Q_1 y la clase *runningGame* ha sido añadida a Q_2, pero que también necesitamos añadir *startup* porque *runningGame* es una clase refinada. Una secuencia de pruebas es una lista de casos de prueba, los cuales han sido obtenidos aplicando transiciones desde uno previo, a excepción del primero de la lista, que viene impuesto por defecto. Una secuencia de longitud tres para nuestro ejemplo podría estar compuesta por los siguientes casos de prueba: ($Q_1 = \{startingGame, running\}$, $Q_2 = \{runningGame, startup, running\}$ y $Q_3 = \{runningGame, startup, paused\}$).

Si dos transiciones diferentes se pueden usar para transitar y afectan a conjuntos disjuntos de clases, entonces es posible realizar estas transiciones a la vez, y por tanto considerarlas como un solo paso. En nuestro ejemplo, las transiciones *startingGame* → *runningGame* y *running* → *paused* afectan a diferentes clases y pertenecen a clasificaciones hermanas. Por tanto, podemos combinar las transiciones y obtener una secuencia de longitud dos (Q_1,Q_3). Esta secuencia tiene la misma cobertura (de clases y transiciones) que la secuencia (Q_1, Q_2, Q_3).

Dada una secuencia de pruebas definimos la cobertura de clases como el número de clases no repetidas que aparecen en los casos de prueba de una secuencia dividida por el total de clases del modelo ECTM. Definimos la cobertura de transiciones como el número de transiciones no repetidas efectuadas frente al número total de transiciones posibles en el ECTM. El problema que estamos interesados en solucionar consiste en encontrar un conjunto de secuencias tales que la cobertura (clases o transiciones) sea maximizada y la longitud de la secuencia sea minimizada.

IV. Algoritmos de Resolución

En esta sección describimos brevemente los enfoques usados para resolver el TSGP. Primero vamos a presentar un enfoque evolutivo, concretamente un algoritmo genético. En segundo lugar, describimos una propuesta basada en colonias de hormigas. Por último, describimos una técnica voraz propuesta en la literatura para comparar con ella. Quisiéramos destacar que el tamaño de los casos de prueba que componen las secuencias es variable debido a la estructura jerárquica del modelo.

A. Generador Genético de Secuencias de Pruebas

El Generador Genético de Secuencias de Pruebas (GGSP) construye un conjunto de pruebas teniendo en cuenta las dependencias entre los casos de prueba

en la generación de una secuencia de pruebas. GGSP es un algoritmo que optimiza la secuencia por partes, siendo su núcleo un algoritmo que evoluciona una población de soluciones en cada iteración hasta conseguir la máxima cobertura posible. Este bucle es repetido hasta que el criterio de cobertura se satisfaga. El algoritmo intenta encontrar el conjunto de casos de prueba que maximicen la cobertura, para después añadirlos secuencialmente a la solución (secuencias de pruebas).

En esta sección nos vamos a limitar a describir las particularidades de nuestra implementación aplicada al TSGP. Como entradas, el algoritmo necesita el modelo ECTM, los parámetros del algoritmo genético (población, probabilidad de mutación, etc.) y el criterio de cobertura deseado. Al comienzo, la secuencia se inicializa con el caso de prueba inicial (clases por defecto), y el conjunto de cobertura se inicializa con todas las clases o transiciones posibles. En cada iteración del bucle externo, se lanza una instancia del algoritmo genético para que optimice una parte de la secuencia, una vez que hemos optimizado esta parte, la siguiente iteración del algoritmo comenzará desde el último caso de prueba añadido a la secuencia.

La representación que utilizamos para una solución es un vector de enteros de longitud l. Determinamos la longitud de los cromosomas como un parámetro del algoritmo.

$$sol = [I_1, I_2, I_3, ..., I_l].$$

Las transiciones desde la clase actual pueden ser enumeradas, por tanto cada número (I_i) es la siguiente transición seleccionada para transitar de la clase actual a otra.

La evaluación de una solución se hace tomando secuencialmente cada una de las transiciones de la solución, y así generando una secuencia de casos de prueba con una cobertura propia. La función de evaluación selecciona una clase hija c (de izquierda a derecha) y un gen de la solución que se consume para seleccionar la siguiente transición t_i. Entonces, t_i se añade al conjunto de transiciones seleccionadas, T'. En cada transición, la función de evaluación consume, como mucho, tantos genes como clases hoja están presentes en el caso de prueba origen. Puede que necesitemos consumir un número variable de genes de la solución para transitar al siguiente caso de prueba. Esto depende del caso de prueba origen. Usamos la siguiente expresión para seleccionar la siguiente transición:

$$t_i = I_i \mod |Transiciones(c)|. \quad (1)$$

donde I_i es la i componente de la solución y $Transiciones(c)$ es la lista de todas las posibles transiciones con origen en la clase c.

Una particularidad relevante de nuestra implementación de este algoritmo genético es que no usamos el operador de recombinación. Esto es debido a que el intercambio de genes da lugar a secuencias con transiciones sin sentido, ya que el origen de las transiciones es diferente. La interpretación de un gen depende de la clase actual, y a su vez de los genes anteriormente consumidos, por esta razón el uso de la recombinación no beneficia a la búsqueda.

Respecto al operador de mutación, este itera sobre todos los genes de la solución, mutando su valor de forma uniforme con probabilidad pm_1. El operador aumenta esta probabilidad de forma lineal dependiendo de la longitud de la solución hasta un valor pm_2. Con este comportamiento queremos que se mute con menor probabilidad las posiciones iniciales de la solución que deben mantenerse estables, e incrementar esta probabilidad para las posiciones finales de la solución. En este trabajo hemos usado $pm_1 = 0,05$ y $pm_2 = 0,25$

Una posible amenaza a la validez de la experimentación es el sesgo introducido por los parámetros de los algoritmos. No obstante, hemos realizado experimentos previos para seleccionar los mejores parámetros para el algoritmo GGSP. Hemos probado todas las combinaciones de valores mostradas en la Tabla I, siempre con 100.000 evaluaciones. Los parámetros usados como definitivos están marcados en negrita.

TABLA I: Parámetros para GGSP. Los valores de los parámetros usados en la experimentación final están marcados en negrita.

Parámetro	**Valor**
Población	**4**, 8, 10
Cruce	**No**, Si (1,0 , 0,9, 0,8)
Mutación	0,05, 0,1, 0,2, **Adapt. (0,05-0,25)**
Longitud Cromosoma	10, **20**, 50, 100

B. Generador de Secuencias con Colonias de Hormigas

Nuestro algoritmo Generador de Secuencias con Colonias de Hormigas (GSCH) es una adaptación del algoritmo propuesto por Alba y Chicano [13] para grafos de tamaño desconocido. A continuación vamos a destacar las diferencias entre el algoritmo original y nuestra adaptación. Primero, el ACO tradicional busca un camino entre un conjunto inicial de nodos hasta un conjunto final. Como nuestro objetivo en el TSGP es cubrir todas las posibles clases o transiciones, estamos interesados en visitar todas las clases usando todas las posibles transiciones desde un caso de prueba inicial hasta una caso de prueba final. Segundo, GSCH no define clases o casos de prueba finales, el algoritmo añade un nuevo caso de prueba hasta cumplir con el criterio de cobertura.

El algoritmo inicializa todos los rastros de feromonas con un valor entre 0,1 y 10. Todas las hormigas van a partir del caso de prueba inicial o por defecto. Después de la inicialización, el algoritmo entra en un bucle que se ejecuta hasta que se alcanzan un máximo de pasos o la máxima cobertura. En el bucle, cada hormiga selecciona el siguiente nodo de

acuerdo a las feromonas (τ_{ij}) y la heurística (η_{ij}) de esa transición (i, j). La hormiga selecciona un nodo $j \in T(i)$, siendo $T(i)$ el conjunto de todas las posibles transiciones desde las clases del caso de prueba actual, con la siguiente probabilidad:

$$p_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{s \in T(i)}[\tau_{is}]^{\alpha}[\eta_{is}]^{\beta}} \text{ , para } j \in T(i) \text{ ,} \quad (2)$$

donde α y β son dos parámetros del algoritmo que determinan la influencia de los rastros de feromona y la heurística en el camino en construcción.

La función heurística η depende de las transiciones, así que a mayor valor de η_{ij}, mayor probabilidad de tomar la transición (i, j). Usamos la misma heurística que el algoritmo voraz (Sección IV-C), que presentamos en la siguiente sección.

La fase de construcción de la secuencia termina cuando las hormigas llegan a la longitud máxima λ_{ant}, o se cumple el criterio de cobertura. Cuando todas las hormigas han construido sus caminos, se produce la actualización de feromonas. Para más detalles sobre esta actualización, ver [13].

Mostramos en la Tabla II todas las combinaciones de parámetros que hemos probado. Los parámetros usados definitivamente en la comparación están marcados en negrita. El parámetro λ_{ant} lo fijamos a 400 para que permitiese caminos suficientemente largos para alcanzar nuestro objetivo.

TABLA II: Parámetros para GSCH. Los valores de los parámetros usados en la experimentación final están marcados en negrita.

Parámetro	**Valor**
α	**1**, 2, 5
β	1, **2**, 5
ρ	0,1, **0,5**, 0,9
Iteraciones	10, 20, 50, **100**
Tamaño Colonia	2, 5, **10**

C. Algoritmo Voraz Determinista

Este algoritmo fue presentado por primera vez en [9]. Lo usamos en este trabajo para validar los resultados obtenidos por nuestras propuestas bio-inspiradas contra una técnica consolidada en este problema. Este enfoque usa un sistema multi-agente con dos tipos de agentes diferentes que transitan por el ECTM: Un agente llamado *walker agent* y otro llamado *coverage agent*. Los agentes viajan por el modelo de manera que sólo se generan caminos válidos hasta que se obtiene la cobertura deseada. Para una descripción detallada del algoritmo, remitimos al lector a su descripción original en [9].

V. Experimentos

En esta sección de experimentos usamos un *benchmark* con 12 modelos diferentes de programas/artefactos extraídos de la literatura, y que se encuentran disponibles en repositorios como *IBM Rhapsody* ó *Matlab Simulink Stateflow*. Queremos destacar que la mayoría de las instancias son jerárquicas y concurrentes, lo que significa que la longitud de los casos de prueba va a ser variable, lo que es una dificultad añadida. Los detalles de los modelos se encuentran en la Tabla III. La segunda y tercera columna lista el número de clases y transiciones del modelo, mientras que la cuarta y quinta columna nos especifica el número de casos de prueba requeridos para cumplir con el criterio de cobertura mínima y completa para una generación de casos de prueba convencional.

TABLA III: Características generales del *benchmark* de programas.

Nombre	Clases	Trans.	Min.	Completa
Keyboard [14]	5	8	2	4
Microwave [15]	19	23	7	56
Autoradio [16]	20	35	11	66
Citizen [17]	62	74	31	3121
Coffee Machine	21	28	9	81
Communication	10	12	7	7
Elevator	13	18	5	80
Tetris	11	18	10	10
Mealy Moore	5	11	5	5
Fuel Control	5	27	5	600
Transmission	7	12	4	12
Aircraft	24	20	5	625

A. Parámetros de los Experimentos

GSCH and GGSP son algoritmos no-deterministas, así que realizamos 30 ejecuciones independientes para cada combinación programa/tipo cobertura con el objetivo de que el análisis estadístico sea significativo. Hemos aplicado la prueba *Wilcoxon rank-sum* [18] para comprobar si las diferencias entre los algoritmos son estadísticamente significativas. Destacamos el dato en las tablas cuando estas diferencias existen. Hemos fijado un nivel de confianza del 99,9 % (p-valor bajo 0,001) para la comparación completa. Hemos marcado un resultado en gris oscuro cuando es el mejor, y en gris claro cuando es el segundo mejor. Cuando el resultado de un algoritmo es significativamente mejor que otro (típicamente el algoritmo cuyos resultados son más lejanos), añadimos un asterisco. Añadimos dos asteriscos si el resultado es significativamente mejor que los otros dos resultados. Además, con el objetivo de realizar una apropiada interpretación de los resultados de las pruebas estadísticas, siempre es aconsejable informar sobre las *effect size measures*. Para este propósito, hemos usado el estadístico $\hat{A}_{12}$ propuesto por Vargha y Delaney [19]. Esta prueba nos proporciona información acerca de la magnitud de un efecto, el cual puede ser útil para determinar si la significancia estadística tiene importancia práctica. Dada una medida de rendimiento M, $\hat{A}_{12}$ mide la probabilidad de que ejecutando el algoritmo A se alcancen valores mayores para M que ejecutando un algoritmo B. Si estos dos

algoritmos son equivalentes, entonces $\hat{A}_{12} = 0,5$. Un $\hat{A}_{12} = 0,3$ significa que obtendremos valores más altos usando el algoritmo A en un 30 % de las ocasiones.

La notación usada en las tablas de resultados es como sigue. Un número simple n indica el tamaño de una única secuencia: es el número de casos de prueba generados para obtener el 100 % de cobertura. En el caso de las técnicas bio-inspiradas indicamos la media de las 30 ejecuciones. Un número n seguido de un porcentaje (p %) indica el número de casos de prueba necesarios para alcanzar ese porcentaje de cobertura. Si se indica otro número entre paréntesis (m), se refiere al número de secuencias necesarias para alcanzar esa cobertura. Es posible que exista más de una secuencia de pruebas debido a que existen clases sin transición de salida, lo que exige iniciar otra secuencia para completar la cobertura.

Todas las ejecuciones fueron realizadas en un *cluster* de 16 máquinas con procesadores Intel Core2 Quad (4 núcleos por procesador) a 2.66 GHz y 4 GB de memoria corriendo un S.O. Ubuntu 12.04.1 LTS y gestionado por HT Condor 7.8.4 cluster manager.

B. Análisis de los Experimentos

En esta sección analizamos el comportamiento de los enfoques con el objetivo de destacar el algoritmo que se comporta mejor. Los resultados de las ejecuciones de los algoritmos para cobertura de clases y de transiciones se muestran en la Tabla IV y Tabla VI, respectivamente.

TABLA IV: Resultados de la generación de secuencias de pruebas para cobertura de clases.

Nombre	GGSP	GSCH	Voraz
Keyboard [14]	2	2	2
Microwave [15]	8*	8*	9
Autoradio [16]	13.30*	14	13*
Citizen [17]	39.47*	36**	47
Coffee Machine	9	9	9
Communication	7	7	7
Elevator	6	6	6
Tetris	12*	12*	15
Mealy Moore	5	5	5
Fuel Control	5	5	5
Transmission	4	4	4
Aircraft	4 *(86.20 %)*	4 *(86.20 %)*	4 *(86.20 %)*

Para la cobertura de clases, la cobertura total se alcanzó en 11 de 12 programas. El programa *Aircraft* es el único que resulta en coberturas menores al 100 %, obteniéndose un 86.2 % de cobertura. Esto es debido a que existen clases aisladas en el modelo. En todos los programas se utiliza una sola secuencia, lo cual es deseable. Con respecto a las diferencias existentes, estas se producen en cuatro programas (*Microwave, Autoradio, Citizen*, y *Tetris*). El enfoque voraz obtiene mejores resultados en el programa *Autoradio*, donde la diferencia con GGSP no es significativa. Para los otros tres programas las técnicas bio-inspiradas alcanzan cobertura total con menos casos de prueba. Por ejemplo, ambas reducen el tamaño del conjunto de pruebas en más de un 20 % en el programa *Tetris*.

El análisis del programa *Citizen* es especialmente interesante porque es el más complejo de todos, y en consecuencia las diferencias entre los algoritmos son mayores. GSCH obtiene los mejores resultados en este programa, reduciendo el tamaño del conjunto de pruebas en más del 23 % con respecto al algoritmo voraz, y en un 9 % con respecto a GGSP. Además, GGSP es un 15 % mejor que el algoritmo voraz. Podemos decir que GSCH es el enfoque más preciso y efectivo para el modelo más grande de este estudio.

A la luz de estos resultados y con la intención de determinar si los resultados tienen importancia práctica, analizamos el estadístico $\hat{A}_{12}$. En la Tabla V resumimos la media de los valores del estadístico $\hat{A}_{12}$ para la cobertura de clases y todos los programas. En este caso, el algoritmo A es el que aparece en las filas y el B en las columnas. La diferencia entre los algoritmos no son muy grandes debido a que hemos seleccionado instancias de programas pequeñas, medianas y grandes. Consecuentemente es difícil obtener grandes diferencias en instancias pequeñas y medianas. Numéricamente los resultados de GSCH van a ser mejores que los obtenidos con GGSP y el algoritmo voraz en un 51,39 % y 58,33 % de las ocasiones, respectivamente. Además, los resultados de GGSP van a ser mejores que el algoritmo voraz en un 61,11 % de las ocasiones, lo cual es una gran diferencia.

TABLA V: Resultados de la prueba estadística $\hat{A}_{12}$ para cobertura de clases. A es el algoritmo de la fila y B el de la columna.

	GGSP	GSCH	Voraz
GGSP	-	0.5139	0.3889
GSCH	0.4861	-	0.4167
Voraz	0.6111	0.5833	-

Con respecto a la cobertura de transiciones (Tabla VI), sólo GSCH consigue obtener cobertura total en todos los programas. Los otros dos algoritmos fallan en el programa *Citizen*. Los resultados consisten en una única secuencia de pruebas en 11 de 12 programas, mientras que en el programa *Aircraft* se necesitan dos secuencias. En este caso las diferencias existen en cinco programas (Autoradio, Citizen, Coffee, Communication, and Fuel Control). El algoritmo voraz sólo es mejor que los otros algoritmos en el programa *Coffee Machine*, donde reduce el tamaño de la secuencia. Nuevamente, las diferencias son pequeñas en la mayoría de programas excepto en *Citizen*, donde GSCH es claramente el mejor algoritmo. GSCH es el único algoritmo que siempre consigue el 100 % de cobertura de transiciones. El algoritmo voraz no consigue nunca alcanzar la cobertura total, mientras que GGSP obtiene el 100 % de cobertura la mayoría de ejecuciones. GSCH es mejor que GGSP tanto en

cobertura como en el tamaño del banco de pruebas. GSCH es capaz de reducir el tamaño de las pruebas en un 14,7 % con respecto a GGSP.

TABLA VI: Resultados para la generación de secuencias de pruebas para cobertura de transiciones.

Nombre	GGSP	GSCH	Voraz
Keyboard [14]	5	5	5
Microwave [15]	17	17	17
Autoradio [16]	36.30	36	36
Citizen [17]	75.27* *(99.90 %)*	64.17**	51 *(92.70 %)*
Coffee Machine	19	19	18**
Communication	16*	16*	17
Elevator	9	9	9
Tetris	31	31	31
Mealy Moore	24	24	24
Fuel Control	11*	11*	12
Transmission	9	9	9
Aircraft	7 (2)	7 (2)	7 (2)

La Tabla VII muestra los resultados del estadístico $\hat{A}_{12}$ para cobertura de transiciones. Hemos considerado todos los programas, con la excepción del programa *Citizen* donde los resultados no son comparables. Este hecho es debido a que tanto el algoritmo voraz como GGSP no son capaces de alcanzar la cobertura total de transiciones, resultando en secuencias de pruebas más cortas pero con peor calidad (menor cobertura). Aunque no hemos incluido los resultados de Citizen, donde el algoritmo GSCH es claramente superior, GSCH es mejor que GGSP y el algoritmo voraz en un 51,25 % y 54,55 %, respectivamente. Además, GGSP obtiene secuencias de pruebas más pequeñas que el algoritmo voraz en un 53,29 % de las ocasiones. Con respecto a la calidad de las soluciones (nivel de cobertura), los enfoques bio-inspirados (GSCH y GGSP) parecen competitivos para este problema, ya que son capaces de generar secuencias de pruebas con cobertura máxima, y obtienen mejores resultados que el algoritmo voraz con mayor probabilidad.

TABLA VII: Resultados de la prueba estadística de Vargha y Delaney, ($\hat{A}_{12}$) para cobertura de transiciones. A es el algoritmo de la fila y B el de la columna.

	GGSP	GSCH	Voraz
GGSP	-	0.5125	0.4670
GSCH	0.4875	-	0.4545
Voraz	0.5329	0.5455	-

C. Cobertura vs Tamaño

El incremento del tamaño del conjunto de pruebas para obtener cobertura total es otro aspecto que tenemos que tener en cuenta. Este comportamiento requiere un análisis más profundo para evaluar el compromiso entre cobertura y tamaño, ya que esto es un aspecto crucial al generar pruebas [20]. Ilustramos este comportamiento con el programa *Citizen* en las Figuras 2 y 3 para cobertura de clases y transiciones. En las figuras mostramos la solución determinista del algoritmo voraz y la mediana y rango intercuartílico de las 30 ejecuciones de los algoritmos no-deterministas. Debemos destacar que este análisis se realiza sobre las soluciones ya computadas.

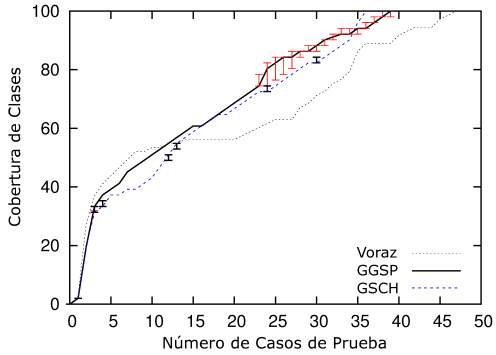

Fig. 2: Solución mediana y rango intercuartílico de GSCH, GGSP y voraz para cobertura de clases.

Vamos a comenzar con el análisis de las soluciones donde queremos obtener cobertura total de clases. En la Figura 2 podemos ver que la cobertura es similar en los primeros casos de prueba. El algoritmo voraz es ligeramente mejor con un 54 % de cobertura. Desde aquí, ambas técnicas bio-inspiradas continúan añadiendo cobertura con el mismo ratio, al contrario que el algoritmo voraz que empeora hacia la mitad de la secuencia. GGSP obtiene su ventaja máxima cuando alcanza el 80 % de cobertura, mientras que GSCH sólo alcanza un 72 % con los mismos casos de prueba (24). Cuando aún permanecen sin visitar algunas clases, GSCH es capaz de visitarlas en menos pasos. Por tanto, alcanza la cobertura total en 36 pasos, 3 menos que GGSP y 11 menos que el algoritmo voraz.

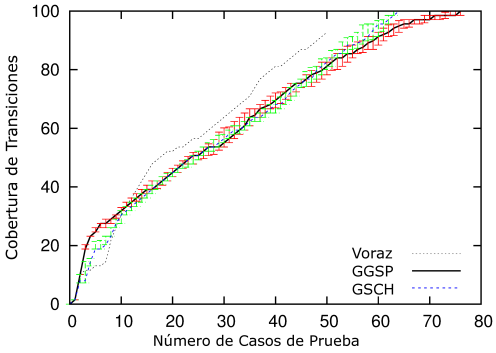

Fig. 3: Solución mediana y rango intercuartilico de GSCH, GGSP y voraz para cobertura de transiciones.

En la Figura 3 mostramos el mismo tipo de gráfico para la cobertura de transiciones. En este caso, GGSP es mejor al comienzo porque es capaz de explorar primero una zona con alta densidad de transiciones, es decir, el algoritmo es capaz de visitar nodos sin tener que tomar varias veces una misma transición. Por su parte, el algoritmo voraz obtiene mejor cobertura usando el mismo número de pruebas desde 12 pruebas en adelante, pero no es capaz de alcanzar más de un 92,7 % de cobertura. En este

caso, GSCH es capaz de alcanzar la cobertura total con menos pruebas, y añade cobertura de forma progresiva con cada caso de prueba añadido. Este gran esfuerzo en reducir los casos de prueba repetidos hace que el algoritmo sea razonablemente predecible. Este comportamiento es deseable porque la cobertura obtenida es proporcional al número de casos de prueba empleados.

VI. Conclusiones

En este artículo hemos extendido el enfoque CTM para hacer posible la generación de secuencias de pruebas. Los beneficios de generar secuencias son claros, ahorramos costes y tiempo al ejecutar las pruebas de forma secuencial, ya que el caso de prueba previo sitúa al SUT en el estado adecuado para probar la siguiente funcionalidad.

En este trabajo hemos comparado nuestros resultados con un algoritmo voraz determinista existente en la literatura. Hemos ejecutado los tres algoritmos para generar secuencias de pruebas para 12 programas diferentes y para cobertura de clases y transiciones. Después de analizar las soluciones obtenidas por los tres enfoques, podemos concluir que las técnicas bio-inpiradas son significativamente mejores que el algoritmo voraz en el programa más complejo, especialmente GSCH. El algoritmo voraz sólo es mejor que GGSP y GSCH en uno y dos escenarios de ocho, donde existen diferencias significativas. GGSP es significativamente mejor que el algoritmo voraz en cuatro de los ocho escenarios. Finalmente, GSCH es mejor que el algoritmo voraz en seis de ocho escenarios donde existen diferencias significativas. En consecuencia, GSCH es el mejor algoritmo de la comparación. Tiene un buen compromiso entre el número de casos de prueba de la secuencia generada y su cobertura.

La investigación futura va enfocada a enfrentarse a criterios de cobertura t-wise con valores de $t \geq 2$ para la generación de secuencias. En otras palabras, necesitamos algoritmos eficientes para ser capaces de computar, al menos, cobertura de pares para clases y transiciones. Esto va a suponer un incremento exponencial en el tamaño de las secuencias de pruebas para cumplir un criterio tan estricto, aunque la cobertura de pares añadirá más confianza en la fase de pruebas. Finalmente, aunque hemos obtenido muy buenos resultados con el enfoque basado de colonias de hormigas, queremos explorar el uso de algoritmos de trayectoria como *Simulated Annealing* que han obtenido buenos resultados en pruebas combinatorias [21].

Agradecimientos

Esta investigación ha sido financiada por el Ministerio de Economía y Competitividad, y fondos FEDER con el contrato TIN2011-28194 (proyecto roadME), la Universidad Técnica de Ostrava con contrato OTRI 8.06/5.47.4142, el proyecto europeo FITTEST Project (EU FP7-ICT-257574), la beca BES-2012-055967, la Universidad de Málaga y Andalucía Tech.

Referencias

[1] "The economic impacts of inadequate infrastructure for software testing," Tech. Rep., NIST, May 2002.

[2] Elfriede Dustin, *Effective Software Testing: 50 Ways to Improve Your Software Testing*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] Mark Harman, "The current state and future of search based software engineering," in *(ICSE/FOSE '07*, Minneapolis, Minnesota, USA, 20-26 May 2007, pp. 342–357, IEEE Computer Society.

[4] Phil McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.

[5] Tanja Vos, Felix Lindlar, Benjamin Wilmes, Andreas Windisch, Arthur Baars, Peter M. Kruse, Hamilton Gross, and Joachim Wegener, "Evolutionary functional black-box testing in an industrial setting," *Software Quality Changes*, 2012.

[6] Cem Kaner, Copyright Cem, and Kaner All, "The impossibility of complete testing," 1997.

[7] Myra Cohen, Joshua Snyder, and Gregg Rothermel, "Testing across configurations: implications for combinatorial testing," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–9, November 2006.

[8] Alan W. Williams and Robert L. Probert, "A measure for component interaction test coverage," in *Proc. ACS–IEEE Intl. Conf. on Computer Systems and Applications*, 2001, vol. 30, pp. 301–311.

[9] Peter M Kruse and Joachim Wegener, "Test sequence generation from classification trees," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 539–548.

[10] Matthias Grochtmann and Klaus Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993.

[11] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Commun. ACM*, vol. 31, pp. 676–686, June 1988.

[12] Mirko Conrad, "Systematic testing of embedded automotive software - the classification-tree method for embedded systems," in *Perspectives of Model-Based Testing*, 2005, pp. 1–12.

[13] Enrique Alba and Francisco Chicano, "Finding safety errors with ACO," in *GECCO'07*, London, UK, July 2007, pp. 1066–1073, ACM Press.

[14] U Mirosamek, "Two orthogonal regions (main keypad and numeric keypad) of a computer keyboard.," 2009.

[15] Paul J. Lucas, *An Object-Oriented Language System For Implementing Concurrent, Hierarchical, Finite State Machines*, Ph.D. thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1989.

[16] Steffen Helke, *Verifikation von Statecharts durch struktur- und eigenschaftserhaltende Datenabstraktion*, Ph.D. thesis, Technische Universitat Berlin, 2007.

[17] David Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[18] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman & Hall, 2007.

[19] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25(2), pp. 101–132, 2000.

[20] Akbar Siami Namin and James H. Andrews, "The influence of size and coverage on test suite effectiveness," New York, NY, USA, 2009, ISSTA '09, pp. 57–68, ACM.

[21] Jose Torres-Jimenez and Eduardo Rodriguez-Tello, "New bounds for binary covering arrays using simulated annealing," *Information Sciences*, vol. 185, pp. 137–152, 2012.