Prueba de Mutación Evolutiva Aplicada a Sistemas Orientados a Objetos

Pedro Delgado-Pérez[1], Inmaculada Medina-Bulo[1], Sergio Segura[2], Antonio García-Domínguez[3] y Juan José Domínguez-Jiménez[1]

[1] Departamento de Ingeniería Informática, Universidad de Cádiz, España
{pedro.delgado, inmaculada.medina, juanjose.dominguez}@uca.es
[2] Departamento de Ingeniería Informática, Universidad de Sevilla, España
sergiosegura@us.es
[3] Department of Computer Science, University of York, United Kingdom
antonio.garcia-dominguez@york.ac.uk

Resumen A pesar del beneficio que puede reportar la prueba de mutaciones en el proceso de prueba de software, el coste que supone su aplicación siempre ha sido visto como un obstáculo para una mayor acogida por parte de la industria. Por esta razón, se han desarrollado diversas técnicas que tratan de paliar el problema, principalmente mediante la reducción del número de mutantes que son generados. Entre ellas se encuentra la Prueba de Mutación Evolutiva (PME), que propone el empleo de algoritmos evolutivos para encontrar un subconjunto de mutantes que presenta mayor posibilidad de ayudar a refinar el conjunto de casos de prueba empleado. La técnica solo había sido probada con éxito en operadores para el lenguaje de programación WS-BPEL. En este artículo se presentan los experimentos llevados a cabo aplicando la técnica de PME con mutantes generados por operadores de mutación para C++ relacionados con la orientación a objetos. Los resultados obtenidos, usando los parámetros considerados como más apropiados para la configuración del algoritmo, revelan que la técnica también es más efectiva que una estrategia aleatoria con operadores de clase para sistemas en C++.
Palabras clave: Prueba de software, prueba de mutaciones, algoritmos evolutivos, C++, orientación a objetos.

1. Introducción

La *prueba de mutaciones* es una técnica de prueba de software que se emplea tanto para evaluar como para mejorar la habilidad que presenta un conjunto de casos de prueba detectando fallos en el código [22]. La técnica se basa en crear versiones modificadas del programa original, que se conocen como *mutantes*. Cada mutante contiene un simple cambio sintáctico que es insertado en el código mediante los *operadores de mutación* definidos para cada lenguaje de programación. Una vez generados los mutantes, estos son ejecutados contra el conjunto de casos de prueba, al igual que el programa original. Como resultado de este proceso, podemos obtener mutantes que han sido *matados* porque la salida para alguno de los casos de prueba difería de la producida por el programa original, o,

por el contrario, mutantes que permanecen *vivos* porque no se observa ninguna diferencia. Estos últimos pueden ayudarnos a generar nuevos casos de prueba para conseguir detectar esas mutaciones, pudiendo también ocurrir que algunas de las mutaciones no hayan en realidad cambiado la semántica del programa y los mutantes generados sean, por tanto, *equivalentes* al programa original.

Debido a la gran cantidad de mutantes que se pueden derivar de un programa, la prueba de mutaciones es conocida por ser una técnica de alto coste computacional. Este problema unido a la ardua tarea de determinar qué mutantes de los que permanecen vivos son equivalentes suponen una traba para su aplicación. Para reducir el elevado coste que presenta la prueba de mutaciones, se han propuesto diferentes técnicas a los largo de los años, principalmente para reducir el número de mutantes a generar. Entre estas técnicas cabe destacar la mutación selectiva [21], la mutación por muestreo [1], la mutación por agrupamiento [11] o la mutación de orden superior [13]. Una novedosa técnica para la reducción del número de mutantes a generar pero manteniendo un alto potencial para refinar el conjunto de casos de prueba es la conocida como *Prueba de Mutación Evolutiva* [9] (PME de ahora en adelante). Esta técnica se basa en el uso de un algoritmo genético para guiar la búsqueda hacia la obtención de un subconjunto de mutantes útiles para la mejora del conjunto de casos de prueba.

Este artículo tiene como objetivo mostrar los resultados que ofrece la PME en sistemas orientados a objetos en lenguaje C++. La técnica había sido empleada anteriormente con composiciones WS-BPEL con buenos resultados en comparación con el uso de una selección puramente aleatoria. Sin embargo, no se habían llevado a cabo estudios posteriores sobre la aplicabilidad de la PME en diferentes contextos. Los resultados experimentales sobre tres programas reales de diversos tamaños muestran que la técnica obtiene buenos resultados con operadores orientados a objetos para C++, validando su uso en diferentes ámbitos.

El otro objetivo principal de este artículo es presentar la herramienta desarrollada para unir el sistema de mutaciones en C++ [3] con el algoritmo genético implementado en la herramienta *GAmera* [8]. Asimismo, se explican los cambios que han sido llevados a cabo para poner en práctica la técnica debido a las particularidades del lenguaje y los operadores de mutación estudiados.

El resto del artículo se estructura de la siguiente manera. En la siguiente sección se comentan los antecedentes y el trabajo relacionado. En la Sección 3 se muestra el funcionamiento del algoritmo genético y se presentan las peculiaridades de la herramienta desarrollada que conecta el algoritmo genético con el sistema de mutaciones en C++. En la Sección 4 se muestran y discuten los resultados obtenidos en el experimento llevado a cabo. En la última sección, se exponen las conclusiones y el trabajo futuro a realizar.

2. Antecedentes y Trabajo Relacionado

2.1. Prueba de Mutación Evolutiva

La *PME* [9] es una de las técnicas más recientemente presentadas de reducción de mutantes y consecuente reducción del coste computacional. El objetivo

de esta técnica es la de generar tan solo un subconjunto del total de mutantes que contenga un gran porcentaje de los mutantes que realmente permiten refinar el conjunto de casos de prueba utilizado, de manera que no se incurra en una pérdida significativa de la efectividad de la prueba de mutaciones. La obtención de mutantes con capacidad de mejorar el conjunto de pruebas se lleva a cabo a través de un algoritmo genético, el cual, mediante su *función de aptitud*, favorece a los mutantes que los autores de la técnica denominan como *mutantes fuertes*, que pueden ser de dos tipos a su vez:

- **Mutantes potencialmente equivalentes:** Son aquellos que, con el actual conjunto de casos de prueba, permanecen vivos. Estos mutantes pueden o bien sugerir la creación de nuevos casos de prueba para matarlos, o bien pueden resultar finalmente en mutantes equivalentes (condición que solo se conoce tras la inspección de los mismos).
- **Mutantes difíciles de matar:** Son aquellos mutantes que son matados por un único caso de prueba que, además, solo mata a ese mutante.

La PME fue empleada para reducir el coste de la prueba de mutaciones en composiciones WS-BPEL. Para ello, se desarrolló la herramienta *GAmera* [8], que implementaba el algoritmo genético propuesto por los autores. Como resultado del uso de GAmera en las tres composiciones analizadas para alcanzar un porcentaje de mutantes fuertes, se obtuvo en todas ellas una reducción mayor del número de mutantes que si los mutantes fuesen elegidos de manera aleatoria. También se demostró que el impacto del uso del algoritmo en el tiempo global era marginal. Por último, se establecieron ciertos parámetros de configuración para optimizar los resultados del algoritmo genético.

2.2. Prueba de Mutaciones para C++

La prueba de mutaciones ha sido aplicada desde sus inicios a una gran variedad de lenguajes de programación, tales como Java [15], SQL [20] o más recientemente Python [6]. Gran parte de las herramientas desarrolladas y lenguajes de programación a los que se ha aplicado están recogidos en un trabajo de Jia y Harman [12]. Sin embargo, hasta hace unos años no había sido prácticamente abordada en relación a C++. Respecto a este lenguaje, un conjunto de operadores de mutación a nivel de clase fueron definidos en cuanto a características del paradigma de orientación a objetos como la herencia, el polimorfismo y la sobrecarga de métodos [4,5]. Posteriormente se establecieron unas pautas para la correcta generación de mutantes a la hora de implementar los operadores de mutación [2]. Finalmente, se presentó la herramienta que aplicaba un subconjunto de los operadores de clase definidos para este lenguaje [3]. Por su parte, Kusano y Wang [14] también desarrollaron una herramienta de mutaciones enfocada a la concurrencia en aplicaciones multihilo en C++.

El sistema de mutaciones para C++ desarrollado por los autores de este trabajo admite las mismas órdenes de ejecución que MuBPEL para composiciones WS-BPEL [10]. MuBPEL es la herramienta subyacente empleada por GAmera

para el análisis, generación y ejecución de mutantes. El hecho de usar la misma arquitectura permite que podamos reutilizar con mayor facilidad el algoritmo genético implementado en GAmera para aplicarlo a ficheros de código C++.

3. Funcionamiento de la Prueba de Mutación Evolutiva

3.1. Algoritmo Genético

GAmera [8] identifica a un mutante de forma unívoca mediante tres campos (ver Figura 1):

- **Operador**: Código representativo del operador que genera el mutante.
- **Localización**: Orden en el cual el operador de mutación inserta las diferentes mutaciones dentro del código analizado.
- **Atributo**: Orden en el cual el operador inserta las diferentes mutaciones en una misma localización.

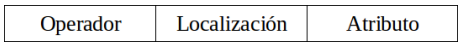

Figura 1. Codificación de un mutante

El algoritmo genético implementado produce una primera generación de individuos (mutantes) aleatoriamente. El tamaño de la población en cada generación viene dado por un parámetro a establecer al inicio. A partir de ahí, produce sucesivas generaciones en las que los nuevos individuos provienen de dos vertientes:

1. **Individuos generados aleatoriamente:** Tal y como sucede en la primera generación, se produce un porcentaje de individuos aleatorios que viene marcado por un parámetro a configurar antes de la ejecución.
2. **Individuos producidos por mutación o por cruce:** El resto de individuos de la generación se producen por esta rama. Se seleccionan de la anterior generación los individuos necesarios mediante el sistema de selección por ruleta (proporcionalmente a la función de aptitud del individuo):
   - **Mutación:** Se muta uno de los campos (operador, localización o atributo) del mutante actual para generar uno nuevo. La mutación del campo realizada se limita para producir un mutante que pueda ser generado. Por ejemplo, si un operador genera mutantes en tres localizaciones, no es posible mutar la localización a cuatro.
   - **Cruce:** Se seleccionan dos individuos que actúan como padres y se selecciona un punto de cruce. Si el punto de cruce se establece entre los campos operador-localización, los padres intercambian el operador manteniendo los campos localización-atributo. Si, por el contrario, el punto de cruce está entre localización-atributo, los padres intercambian el atributo (ver Figura 2).

La probabilidad con la que la herramienta utiliza mutación o cruce también es configurable. Más adelante, en la Sección 3.2, veremos que será necesario modificar estos operadores.

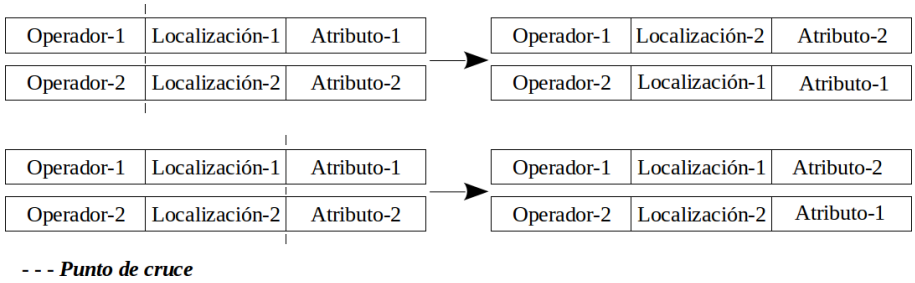

Figura 2. Cruce de mutantes

La función de aptitud para un mutante cuenta el número de casos de prueba que lo matan, así como el número de mutantes que esos casos de prueba matan a su vez. Cuanto mayor sea la suma total de ambas cuentas, peor será la función de aptitud asociada al mutante, pues los casos de prueba que se pueden llegar a generar con ese mutante resultan poco específicos. De esta manera, respecto a los mutantes fuertes (ver Sección 2.1):

- Los mutantes potencialmente equivalentes reciben el valor más alto al no ser matados por ningún caso de prueba.
- Los mutantes difíciles de matar reciben el segundo mejor valor, al ser matados por un solo caso de prueba.

Por el contrario, los mutantes que son matados por todos los casos de prueba son los que reciben las valoraciones más bajas de la función de aptitud. Además de lo dicho, es importante destacar las siguientes características del algoritmo:

- **Uso de segunda población:** El algoritmo utiliza los mutantes generados hasta el momento para estimar de manera más precisa el valor de la aptitud de cada individuo, de manera que la aptitud de un mutante puede variar de una generación a otra.
- **Normalización de valores:** Dado que cada operador genera un número de mutantes distintos en el sistema, para que todos los mutantes tengan la misma probabilidad de ser seleccionados, el sistema normaliza/desnormaliza los valores de los campos localización y atributo.

3.2. Prueba de Mutación Evolutiva para Sistemas Orientados a Objetos en C++

La prueba de mutaciones es una técnica de caja blanca que requiere de marcos de trabajo particulares para cada lenguaje de programación. En este sentido, se

ha desarrollado una herramienta que permite aplicar la PME a aplicaciones codificadas en lenguaje C++. El principal cometido de esta herramienta es la de coordinar la acción entre otros dos sistemas existentes:

- **GAmera** [8]: Esta aplicación implementa el algoritmo evolutivo propuesto por los creadores de la técnica [9].
- **Sistema de mutaciones para C++** [3]: Esta herramienta aplica la prueba de mutaciones al lenguaje C++ mediante operadores a nivel de clase [5].

GAmera fue utilizada para la obtención de resultados experimentales con composiciones en lenguaje WS-BPEL. No obstante, la herramienta está modularizada de manera que el algoritmo genético es independiente del lenguaje al que se quiera aplicar. De esta manera, la herramienta que se ha desarrollado hace corresponder los datos entre GAmera y el sistema de mutaciones para que la conexión entre estos dos sistemas sea posible. La nueva herramienta lleva a cabo las siguientes acciones:

- Transformar las órdenes de ejecución que GAmera requiere para su funcionamiento (análisis, generación y ejecución de mutantes) en órdenes entendibles para el sistema de mutaciones en C++.
- Traducir la salida generada por el sistema de mutaciones al formato de entrada para GAmera y, al contrario, formatear la salida de GAmera como entrada del sistema de mutaciones.

Debido a las características de los dos sistemas a los que esta herramienta conecta, ha sido necesario llevar a cabo varias modificaciones respecto al planteamiento inicial de la técnica. Estos cambios, sin embargo, pueden afectar al funcionamiento del algoritmo genético, tal y como se explica a continuación:

1. Como se comentó en la sección anterior, GAmera identifica a un mutante mediante tres campos: operador, localización y atributo. El campo atributo depende de cada operador de mutación concreto. En este sentido, podemos encontrar las siguientes opciones:
   - **Atributo fijo**: El número de posibles mutaciones a insertar es conocido de antemano. Por ejemplo, un operador que reemplaza un operador aritmético puede reemplazar el operador por un conjunto de operadores aritméticos conocido.
   - **Atributo variable**: El número de posibles mutaciones depende por completo de la localización exacta dentro del código. A modo de ejemplo, un operador de mutación que reemplaza una llamada a un método por otro método que sea compatible (a fin de que no genere un mutante inválido) dependerá del número de métodos compatibles que existan.

   Cuando el algoritmo genético elige el campo "atributo" de un individuo para ser mutado, es necesario que el atributo sea fijo para que el algoritmo pueda seleccionar con igualdad de posibilidades el campo atributo que tendrá el nuevo individuo generado. Es por ello que, cuando un operador de mutación presenta un atributo variable, el atributo se marca siempre como "1" y las diferentes variaciones que se pueden insertar en esa localización se contabilizan simplemente como nuevas localizaciones.

Dicho esto, los operadores de clase implementados en el sistema de mutaciones para C++ o bien solo tienen la posibilidad de generar un único mutante por localización o bien su atributo es variable. Esto quiere decir que, en caso de que el algoritmo genético elija el campo atributo para ser mutado y generar un nuevo individuo, como el campo atributo siempre tiene valor "1", nunca se generaría un individuo diferente a partir de la mutación de ese campo. Es por ello por lo que, para realizar los experimentos, se ha añadido una nueva opción a GAmera para permitir la posibilidad de que el campo atributo no sea mutado para una ejecución dada. Esto también se aplica a la mutación por cruce que intercambia el atributo.

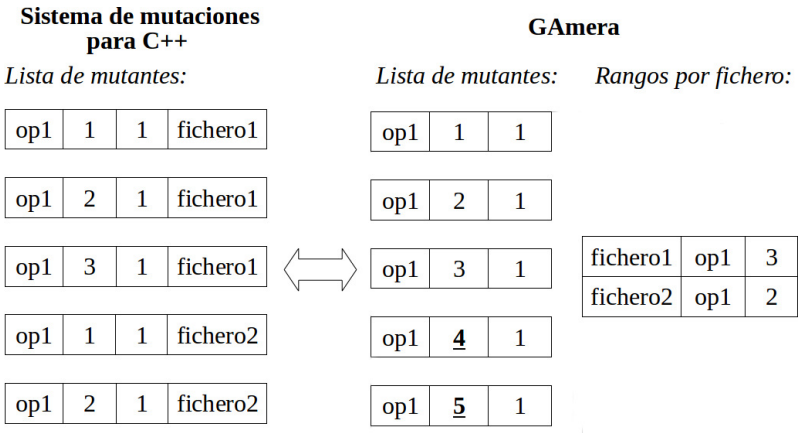

Figura 3. Ejemplo de traducción de mutantes entre el sistema de mutaciones para C++ y GAmera de un operador "op1" aplicado a dos ficheros "fichero1" y "fichero2".

2. Las composiciones WS-BPEL que recibe la herramienta MuBPEL [10] están codificadas en un único fichero fuente. Sin embargo, las aplicaciones en C++ suelen componerse de varios ficheros. El sistema de mutaciones para C++ permite la posibilidad de mutar más de un fichero en la misma ejecución. Para ello, la herramienta añade un campo adicional a la 3-tupla (operador, localización, atributo) para identificar unívocamente cada mutante: el *nombre del fichero* al que pertenece. GAmera solo está configurado para tratar los campos (operador, localización, atributo), de manera que ha sido necesario realizar una traducción entre la salida del sistema de mutaciones y la entrada de GAmera. Esta traducción consiste en hacer corresponder los mutantes entre ambas herramientas mediante la codificación/decodificación del nombre del fichero a través de la localización. Este proceso se detalla gráficamente en la Figura 3.

   No obstante, el hecho de permitir que más de un fichero de código sea mutado en la misma ejecución produce un efecto en el algoritmo genético no

contemplado en los experimentos originales con respecto a WS-BPEL. Tanto la mutación del campo "operador" como del campo "localización" puede provocar que el nuevo individuo generado corresponda a una clase que esté en un fichero de código diferente al del individuo que fue mutado. Aunque las clases de un mismo programa suelen usar un mismo patrón de diseño y estar interrelacionadas, el comportamiento de un operador en clases diferentes puede variar, principalmente cuando se encuentran en ficheros de código distintos. Por tanto, este hecho puede afectar al algoritmo genético puntualmente, pero también puede suponer una ventaja al permitir al algoritmo abrirse hacia nuevos espacios de búsqueda.

4. Experimentos

4.1. Casos de estudio

Para los experimentos conducidos en este artículo, se ha hecho uso de tres programas reales de código abierto:

1. **Dolphin** [7]: Administrador de archivos en aplicaciones de escritorio de KDE por defecto.
2. **Tinyxml2** [19]: Analizador sencillo y eficiente de código XML.
3. **QTDom** [16]: Módulo de Qt que ofrece una implementación en C++ del estándar DOM.

En cada aplicación, los mutantes que se pueden generar con los operadores de mutación implementados en el sistema de mutaciones son siempre los mismos, es decir, los mutantes no varían en diferentes ejecuciones. Por lo tanto, a partir del resultado de una ejecución previa de todos los mutantes en cada aplicación, es posible conocer el número total de mutantes fuertes existentes. Tal y como puede verse en el Cuadro 1, estas aplicaciones producen un número diferente de mutantes totales así como de mutantes fuertes, lo cual nos permite observar el funcionamiento de la técnica en relación a la cantidad de mutantes que se genera. Además, en esta tabla también puede observarse el tamaño del conjunto de casos de prueba empleado para cada uno de los programas, el cual es el que originalmente viene distribuido con cada aplicación.

Cuadro 1. Mutantes y conjunto de casos de prueba de las aplicaciones del estudio

	Dolphin	**Tinyxml2**	**QtDom**	**Total**
Mutantes totales	219	614	1,146	1,979
Válidos	208	433	681	1,322
Fuertes	103	159	348	610
% Mutantes fuertes	49.5%	36.7%	51.1%	46.1%
Casos de prueba	61	57	46	164

Cuadro 2. Parámetros utilizados en la ejecución del algoritmo evolutivo

Parámetro	**Valor**
Tamaño población	5%
Individuos generados aleatoriamente	10%
Individuos generados por mutación o cruce	90%
- Probalididad de mutación	30%
- Probabilidad de cruce	70%

4.2. Configuración del Experimento

El experimento llevado a cabo busca analizar el resultado obtenido por la PME en comparación con una técnica aleatoria:

1. **Aleatoria:** Consiste en ordenar aleatoriamente los mutantes al inicio y después ir seleccionándolos de uno en uno hasta llegar a la condición de parada.
2. **PME:** A partir de la población inicial, se irán produciendo nuevas generaciones de mutantes en base a los parámetros establecidos para el funcionamiento del algoritmo evolutivo. Los parámetros empleados son los que se pueden observar en el Cuadro 2, los cuales son los que en los experimentos realizados en el artículo en el que se presenta la técnica fueron hallados como los más adecuados [9]. El tamaño de la población se refiere al número de mutantes que serán producidos en cada generación, el cual es un porcentaje respecto del total de mutantes en cada aplicación. Como puede observarse, la suma del porcentaje de mutantes generados aleatoriamente y por mutación/cruce es de 100 %, pues a través de estas dos vertientes se crean todos los mutantes de una generación.

En este experimento se busca localizar a los mutantes fuertes, por lo cual se han establecido dos condiciones de parada: al llegar al 75 % y al 90 % del total de mutantes fuertes. Como se mencionó anteriormente, este criterio es posible establecerlo ya que, debido a una ejecución previa, conocemos el número de mutantes fuertes existentes. La herramienta ha sido configurada para medir tanto el porcentaje de mutantes generados hasta el momento de llegar a los límites de parada como el tiempo total necesario para la ejecución de los casos de prueba en estos mutantes. Para evitar el sesgo que puede producirse en una única ejecución de las técnicas (debido al componente aleatorio que ambas conllevan), se han realizado 30 ejecuciones con diferentes semillas de partida. De esta manera, los cálculos a mostrar se obtienen a partir de los datos de estas 30 ejecuciones.

Es necesario remarcar que, para saber si un mutante es o no fuerte, es necesaria la ejecución de todos los casos de prueba (por lo que no basta parar la ejecución al primer caso de prueba que mata al mutante). Asimismo, el algoritmo evolutivo utilizará esta información para el cálculo de la función de aptitud.

4.3. Resultados

Los resultados del experimento se recogen en el Cuadro 3 y 4. En el Cuadro 3 se muestra la media, mediana, valores mínimos y máximo y desviación estándar

en cuanto al porcentaje de mutantes que se necesita generar para obtener un 75 y un 90 % de los mutantes considerados como fuertes. En esta tabla se presentan los datos tanto de la PME como de la técnica aleatoria para los tres casos de estudio. Como puede observarse, el uso del algoritmo genético nos conduce a producir un porcentaje menor de mutantes en todos los casos para lograr los porcentajes de mutantes fuertes establecidos (75 % y 90 %). Especialmente destacable es la diferencia en promedio en torno al 10 % conseguida respecto de la técnica aleatoria en *Tinyxml2*. También podemos observar que la diferencia entre ambas técnicas se reduce al llegar a un umbral más alto (en este caso del 90 %). Por ejemplo, refiriéndonos nuevamente a *Tinyxml2*, la diferencia se acorta en algo más de 4 puntos porcentuales.

En el Cuadro 4 se indica el tiempo en minutos que tardó la ejecución total de cada caso de prueba contra cada mutante generado antes de llegar a la condición de parada. Al igual que en el cuadro anterior, se calcula la media, mediana, mínimo, máximo y desviación estándar. Hay que hacer notar en este punto que los tiempos mostrados para cada una de las aplicaciones varían debido a que cada una de ellas es estudiada junto con un conjunto de casos de prueba, cada uno de los cuales toma un tiempo distinto en ejecutarse. El tiempo que tarda la PME es menor en todos los casos en los distintos parámetros estadísticos mostrados. No obstante, excepto en *Tinyxml2* y límite *75 %*, la desviación estándar es menor usando aleatoriedad que la técnica evolutiva.

4.4. Análisis y discusión de resultados

La PME ha mostrado necesitar generar menos mutantes para conseguir producir una mayor cantidad de mutantes fuertes que la selección aleatoria en los tres casos de estudio analizados. Cuando buscamos un 75 % de los mutantes fuertes, en promedio el ahorro en cuanto al número de mutantes es de alrededor al 6.6 %, siendo destacable el caso de *Tinyxml2* en más de un 10 %. El resultado sigue siendo mejor que la técnica aleatoria cuando se trata de encontrar un porcentaje muy elevado de mutantes fuertes (90 %), aunque la mejora en promedio baja al 3.7 %. Esto puede ser debido en parte a la existencia de unos pocos mutantes fuertes en operadores que en su mayoría generan mutantes considerados de baja calidad para el algoritmo genético. A fin de conocer si estas diferencias son estadísticamente significativas ($p\text{-}value<0.01$), también se ha realizado un estudio estadístico con la aplicación *STATService* [18], que aplica el test apropiado para los datos de las distintas ejecuciones (*Test* en Cuadro 3). En concreto, los test estadísticos aplicados han sido *Wilcoxon* para el caso *Tinyxml2* con umbral 90 % y *T de student* para los otros 5 casos. Los resultados en todos los casos nos llevan a aceptar que la mediana de los porcentajes de mutantes generados por la técnica evolutiva para generar un porcentaje de mutantes fuertes es significativamente menor que la aleatoria. Como es de esperar, el tiempo es también mejor en todos los casos ya que el número de mutantes a ejecutar es siempre menor. El ahorro promedio del tiempo es de 3.3 y 2.7 minutos en la ejecución de los casos de prueba para el umbral del 75 % y del 90 % respectivamente.

Cuadro 3. Porcentaje de mutantes generados por la técnica de PME y la técnica aleatoria hasta conseguir un 75 % y 90 % de los mutantes fuertes.

Umbral	***75%***		***90%***	
Técnica	*PME*	*Aleatoria*	*PME*	*Aleatoria*
Dolphin				
Media	69.87	75.11	85.35	89.07
Mediana	70.09	75.79	85.38	89.72
Mínimo	62.55	67.57	78.99	82.64
Máximo	76.71	81.27	90.41	93.15
D.E.	3.57	3.57	2.67	2.86
Test	p-value	4.55×10^{-07}	p-value	2.72×10^{-06}
Tinyxml2				
Media	64.91	74.93	84.32	89.98
Mediana	64.74	74.83	84.12	90.22
Mínimo	60.58	69.70	77.85	85.01
Máximo	71.49	80.61	89.73	93.64
D.E.	2.59	2.78	3.34	1.78
Test	p-value	7.14×10^{-21}	p-value	1.65×10^{-06}
QtDom				
Media	69.96	74.43	87.84	89.76
Mediana	70.15	74.34	88.09	89.75
Mínimo	66.05	71.64	83.33	86.21
Máximo	73.38	78.88	90.13	93.71
D.E.	1.98	2.00	1.60	1.58
Test	p-value	4.31×10^{-12}	p-value	1.71×10^{-05}

A la vista de los resultados, parece que no hay una gran dependencia del algoritmo genético en relación a la cantidad de mutantes producida en cada caso de estudio. A diferencia de los resultados de los experimentos en [9], el porcentaje más bajo de mejora se obtiene en el caso de estudio que engendra más mutantes. Por lo tanto, de los resultados de este experimento no podemos concluir que la técnica escale en proporción al tamaño del programa analizado, tal y como lo hacen los autores de la técnica en su artículo.

No obstante, analizando los resultados del Cuadro 1, parece que el beneficio reportado por el algoritmo genético está más en relación a la cantidad de mutantes fuertes de partida existentes. Como puede observarse en la tabla, *Tinyxml2* es el que tiene un menor porcentaje de mutantes fuertes (36.8 %), seguido de *Dolphin* (49.5 %) y *QtDom* (51.1 %). De hecho, volviendo a los resultados del Cuadro 3, el porcentaje de mutantes a generar por la técnica evolutiva en promedio también está en consonancia con el orden respecto al porcentaje de mutantes fuertes en estos programas. Por ejemplo, mirando el umbral del 75 %, la técnica evolutiva genera el 64.91 % del total de mutantes para *Tinyxml2*, mientras que requiere de la creación del 69.87 % y 69.96 % de los mutantes para *Dolphin* y *QtDom* respectivamente. Los operadores de mutación a nivel de clase son conocidos

Cuadro 4. Tiempo de ejecución de los mutantes generados por la técnica de PME y la técnica aleatoria hasta conseguir un 75 % y 90 % de los mutantes fuertes.

Umbral	***75%***		***90%***	
Técnica	*PME*	*Aleatoria*	*PME*	*Aleatoria*
Dolphin				
Media	23.78	27.57	29.96	32.39
Mediana	23.54	28.16	29.77	33.15
Mínimo	18.06	19.76	24.81	26.65
Máximo	29.86	31.10	33.75	35.00
D.E.	3.24	2.62	2.36	2.28
Tinyxml2				
Media	19.11	23.61	20.45	24.54
Mediana	18.95	23.60	20.56	24.60
Mínimo	17.08	21.52	18.22	22.80
Máximo	21.56	25.16	21.99	25.34
D.E.	0.96	0.98	1.08	0.59
QtDom				
Media	7.68	9.43	7.84	9.43
Mediana	7.69	9.47	7.82	9.38
Mínimo	6.94	8.71	7.15	8.91
Máximo	8.47	9.89	8.78	9.91
D.E.	0.38	0.29	0.39	0.28

por producir un porcentaje de mutantes equivalentes superior a los operadores tradicionales [17], pero aun así los resultados al comparar ambas técnicas se han mostrado positivos a favor del algoritmo evolutivo también en estos casos.

Por todo lo anterior, concluimos que la PME es una técnica beneficiosa ya que con ella seremos capaces de producir un porcentaje menor de mutantes (consecuentemente, mejorando el coste computacional), aun así conteniendo un elevado porcentaje de los mutantes que nos ayudan a refinar nuestro conjunto de casos de prueba. Los resultados evidencian que cuanto menor es el porcentaje de mutantes fuertes, lo cual suele ocurrir cuando el conjunto de casos de prueba posee ya una capacidad significativa de detección de fallos, mejor serán los resultados de la aplicación de la técnica.

5. Conclusiones y Trabajo Futuro

En este trabajo se presenta una evaluación de la técnica conocida como PME con operadores a nivel de clase para el lenguaje C++. Para ello ha sido necesario desarrollar una aplicación que conectara la herramienta que implementa el algoritmo genético y la que genera los mutantes para C++, introduciendo cambios específicos ajustados a las necesidades del lenguaje y operadores a analizar. Esta técnica de reducción de mutantes, que solo había sido aplicada hasta el momento a composiciones WS-BPEL, ha mostrado ofrecer mejores resultados que la

selección aleatoria de mutantes a la hora de encontrar aquellos mutantes que pueden ayudarnos a mejorar nuestro conjunto de casos de prueba, posibilitando el objetivo de la técnica que es la reducción del coste computacional. El hecho de que la técnica haya demostrado su utilidad para lenguajes y operadores de mutación diferentes, convierte a la PME en una técnica a tener en cuenta para el proceso de mejora de los casos de prueba en cualquier contexto. Como dato novedoso, a la luz de los resultados parece que el rendimiento de la PME respecto a la técnica aleatoria mejora cuanto menor es el porcentaje de mutantes fuertes existentes, y no depende significativamente del número de mutantes total, tal y como indicaban los primeros experimentos en los que se aplicaba la técnica. A pesar de que usando operadores a nivel de clase es probable encontrar un porcentaje elevado de mutantes potencialmente equivalentes, la técnica ha obtenido mejores resultados también en estos casos en nuestros experimentos. No obstante, en nuestros experimentos se han usado tres casos de estudio y sería conveniente confirmar esta tendencia en nuevos experimentos con un número mayor de aplicaciones.

Como trabajo futuro, resultaría interesante repetir estos experimentos para además de calcular el porcentaje de mutantes fuertes, comprobar en qué medida esta técnica nos ayuda realmente a refinar un conjunto de casos de prueba antes que otras técnicas. Asimismo, se podrían aplicar cambios al algoritmo genético para introducir nuevas variables, como la introducción de un porcentaje de los mutantes generados en una generación en la siguiente.

Agradecimientos: Este trabajo fue parcialmente financiado por la beca de investigación PU-EPIF-FPI-PPI-BC 2012-037 de la Universidad de Cádiz, por la Red de Excelencia SEBASENET (TIN2015-71841-REDT), por los proyectos nacionales del Ministerio de Economía y Competitividad DArDOS (TIN2015-65845-C3-3-R) y TAPAS (TIN2012-32273), y por los proyectos de la Junta de Andalucía THEOS (TIC-5906) y COPAS (P12-TIC-1867).

Referencias

1. Budd, T.A.: Mutation Analysis of Program Test Data. Ph.D. thesis, Yale University (1980)
2. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J.: Generación de mutantes válidos en el lenguaje de programación C++. In: XIX Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2014. Cádiz, Spain (2014)
3. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J.: Herramienta para la prueba de mutaciones en el lenguaje C++. In: XX Jornadas de Ingeniería del Software y Base de Datos, JISBD 2015. Santander, Spain (2015)
4. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A.: Operadores de mutación a nivel de clase para el lenguaje C++. In: XII Jornadas sobre Programación y Lenguajes, PROLE 2013. Madrid, Spain (2013)
5. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J., García-Domínguez, A., Palomo-Lozano, F.: Class mutation operators for C++ object-oriented systems. Annals of telecommunications 70(3-4), 137–148 (2015), http://dx.doi.org/10.1007/s12243-014-0445-4

6. Derezinska, A., Halas, K.: Experimental evaluation of mutation testing approaches to Python programs. In: Proceedings of the 9th Workshop on Mutation Analysis (MUTATION'14). pp. 156–164. Cleveland, USA (March 2014), http://dx.doi.org/10.1109/ICSTW.2014.24
7. Dolphin. https://www.kde.org/applications/system/dolphin, [Online; accessed 14-March-2016]
8. Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I.: GAmera: an automatic mutant generation system for WS-BPEL compositions. In: Eshuis, R., Grefen, P., Papadopoulos, G.A. (eds.) Proceedings of the 7th IEEE European Conference on Web Services. pp. 97–106. IEEE Computer Society Press, Eindhoven, The Netherlands (Nov 2009)
9. Domínguez-Jiménez, J.J., Estero-Botaro, A., García-Domínguez, A., Medina-Bulo, I.: Evolutionary mutation testing. Information and Software Technology 53(10), 1108–1123 (Oct 2011), http://dx.doi.org/10.1016/j.infsof.2011.03.008
10. García-Domínguez, A., Estero-Botaro, A., Medina-Bulo, I., Palomo-Lozano, F.: Herramienta de mutación firme para WS-BPEL 2.0. In: Actas de las XVII Jornadas de Ingeniería del Software y Bases de Datos JISBD 2012. pp. 415–418 (2012)
11. Hussain, S.: Mutation Clustering. Master's thesis, King's College London (2008)
12. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on 37(5), 649 –678 (Oct 2011), http://dx.doi.org/10.1109/TSE.2010.62
13. Jia, Y., Harman, M.: Higher order mutation testing. Information and Software Technology 51(10), 1379–1393 (Oct 2009), http://dx.doi.org/10.1016/j.infsof.2009.04.016
14. Kusano, M., Wang, C.: CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. pp. 722–725. IEEE (2013), http://dx.doi.org/10.1109/ASE.2013.6693142
15. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: An automated class mutation system: Research articles. Software Testing, Verification and Reliability 15(2), 97–133 (Jun 2005), http://dx.doi.org/10.1002/stvr.v15:2
16. QtDOM. https://github.com/qtproject/qtbase/tree/dev/src/xml/dom, [Online; accessed 14-March-2016]
17. Segura, S., Hierons, R.M., Benavides, D., Ruiz-Cortés, A.: Mutation testing on an object-oriented framework: An experience report. Information and Software Technology 53(10), 1124–1136 (2011), http://dx.doi.org/10.1016/j.infsof.2011.03.006, special Section on Mutation Testing
18. STATService. http://moses.us.es/statservice, [Online; accessed 20-June-2016]
19. Tinyxml2. https://github.com/leethomason/tinyxml2, [Online; accessed 14-March-2016]
20. Tuya, J., Suárez-Cabal, M.J., la Riva, C.d.: Mutating database queries. Information and Software Technology 49(4), 398–417 (Apr 2007), http://dx.doi.org/10.1016/j.infsof.2006.06.009
21. Wong, W.E., Mathur, A.P.: Reducing the cost of mutation testing: An empirical study. techreport, Purdue University, West Lafayette, Indiana (1993)
22. Woodward, M.R.: Mutation testing - its origin and evolution. Information and Software Technology 35(3), 163–169 (Mar 1993), http://dx.doi.org/10.1016/0950-5849(93)90053-6