Un algoritmo genético para generar casos de prueba funcionales con oráculo

Macario Polo[1], Óscar Pedreira[2], Ignacio García-Rodríguez[1], Nieves R. Brisaboa[2] y Mario Piattini[1]
[1] Escuela Superior de Informática, Universidad de Castilla-La Mancha, Ciudad Real, España
[2] Facultade de Informática, Universidad de A Coruña, A Coruña, España
macario.polo@uclm.es, oscar.pedreira@udc.es, ignacio.grodriguez@uclm.es, brisaboa@udc.es, mario.piattini@uclm.es

Abstract. Muchos autores han utilizado algoritmos genéticos para la generación automática de casos de prueba. En este contexto, la función de fitness se describe dependiendo del objetivo del test suite que se desea obtener, que suele corresponder con algún criterio de cobertura, como cubrir todas las decisiones con MC/DC o cubrir todos los usos de una variable. Uno de los principales problemas de los casos de prueba generados automáticamente es la ausencia del oráculo, que es el mecanismo del que se dota al caso de prueba para que éste determine si ha encontrado o no error en el sistema bajo prueba: en efecto, un caso de prueba obtenido del cruce de dos progenitores estará describiendo un escenario de ejecución no previsto y del que se desconoce su resultado esperado. Por ello, tras la convergencia del algoritmo genético mediante la producción de un test suite que alcanza el requisito de prueba deseado, el tester debe añadir manualmente el oráculo a cada uno de los casos de prueba obtenidos.
En este artículo se presenta un método que permite describir de manera genérica multitud de situaciones de ejecución, así como una estrategia para generar casos que combina expresiones regulares y técnicas combinatorias. Los casos de prueba que se obtienen con esta estrategia conforman la población inicial de un algoritmo genético, que en cada iteración produce una nueva población de casos. Cada uno de estos casos ejercita una de esas situaciones de ejecución genéricas. Como estas situaciones tienen su oráculo descrito también de forma genérica, el test suite que se obtiene está formado por casos de prueba con oráculo. La función de fitness está basada en el Mutation Score que se alcanza al matar mutantes producidos mediante operadores que utilizan programación orientada a aspectos.

Keywords: Generación de casos de prueba, Algoritmos genéticos, Mutación, Oráculo.

1 Introducción

El objetivo principal del testing de software es encontrar errores en el SUT (el *system*

M. Genero, M. Kalinowski (Eds.): CIbSE 2018, Bogotá - Colombia, 2018

under test o sistema bajo prueba). Para ello, el tester escribe casos de prueba que ejercitan tantos escenarios diferentes del SUT como sea posible, teniendo en cuenta las restricciones de coste y tiempo del proyecto. Una propiedad importante de los casos de prueba es que sean reproducibles: es decir, que ofrezcan siempre el mismo resultado (denominado *veredicto* en este contexto) independientemente del momento en que se ejecuten. La reproducibilidad es importante, sobre todo, en metodologías ágiles o iterativas (pues tras cada iteración o *sprint* se reejecutan los casos de prueba procedentes de iteraciones anteriores) y durante las pruebas de regresión en el proceso de mantenimiento (se ejecutan los casos de prueba de versiones anteriores para comprobar que no se han introducido en la nueva *release* errores que antes no existían). En este sentido, un caso de prueba consta en general de tres partes [1][2]: (1) **Especificación de la situación inicial**, que pone al SUT en el estado requerido para ejecutar el caso. Puede incluir, por ejemplo, la creación de una base de datos, el levantamiento de un servidor, etcétera; (2) **Ejecución de las operaciones del SUT** que se desean reproducir, con los datos que correspondan; y (3) **Comparación del resultado** obtenido con el esperado para emitir el veredicto, normalmente un resultado de paso o fallo en función de que, respectivamente, el caso de prueba haya o no encontrado error en el SUT. A este mecanismo de comparación se lo llama *oráculo.*

La automatización de las tareas de testing pretende la ejecución, tan desatendida como sea posible, de sus diferentes fases. De éstas, las más dependientes de la tecnología y del entorno en que se ejecuta el SUT son las siguientes:

1. **Generación de datos y casos de prueba**. Una vez identificados los escenarios que se desean probar, el tester debe identificar las operaciones necesarias para ejecutarlos, así como los datos que se deben pasar a dichas operaciones como parámetros.
2. **Ejecución de casos de prueba**. Los casos de prueba se lanzan contra el SUT. Dependiendo de la plataforma en que corre el SUT, se necesitarán entornos de ejecución de diferentes tipos.
3. **Análisis de resultados**. Además de revisarse los veredictos que hayan arrojado los casos de prueba (y con los cuales se reportan los errores encontrados para que sean corregidos), se realiza también un análisis de cobertura, con el que se conocen las áreas del sistema que han sido recorridas y las que no.

Así pues, en la fase de análisis de resultados se hace tanto una comprobación de los veredictos como de las zonas del código que se han recorrido. De acuerdo con Polo y Reales [3], en un proceso de pruebas en el que se combinan pruebas de caja negra y de caja blanca: (1) se determina inicialmente el nivel de cobertura que se desea alcanzar en el SUT; (2) se construye un test suite para encontrar errores y se analizan los veredictos; (3) si hay errores, el SUT se corrige; (4) si no los hay, se evalúa la cobertura alcanzada: si es igual o superior al umbral preestablecido, se detiene el proceso porque el SUT se está recorriendo suficientemente sin haber encontrado errores; si es inferior, se añaden nuevos casos de prueba para subir la cobertura. Estos nuevos casos de prueba pueden encontrar errores en esas áreas que habían permanecido inexploradas.

Mientras que la generación de datos y casos de prueba para alimentar el test suite inicial son tareas relativamente resueltas, la inclusión automática del oráculo en los casos de prueba, incluso para el test suite inicial, es una tarea complicada que es aún

objeto de investigación. En un proceso de generación, ejecución y análisis automáticos, la situación se complica mucho.

Este artículo describe un proceso automático para generar casos de prueba con oráculos basado en un algoritmo genético, así como una herramienta de soporte. El trabajo está organizado como sigue: en la Sección 2 se presenta un ejemplo ilustrativo que describe el problema objeto de estudio. En la Sección 3 hacemos un breve repaso a algunos trabajos relacionados. La Sección 4 es la principal contribución de este trabajo, presentando el algoritmo genético. Finalmente, la Sección 5 incluye nuestras conclusiones y las líneas de trabajo futuro.

2 Descripción del problema

Supongamos, con fines ilustrativos, la clase *Account* de la Figura 1, que representa una posible cuenta bancaria que, además de un constructor y una operación para conocer el saldo (*getBalance)*, incluye tres operaciones para ingresar (*deposit),* retirar (*withdraw)* y transferir (*transfer)*.

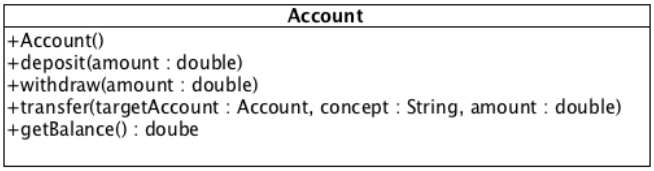

Fig. 1. Una cuenta bancaria

Asumamos que las tres operaciones pueden lanzar las siguientes excepciones: (1) *NegativeAmountException,* si se intenta ingresar, retirar o transferir un importe cero o negativo; (2) *InsufficientBalanceException,* si se intenta retirar o transferir un importe superior al saldo de la cuenta; y (3) *NullTargetAccountException,* si la cuenta de destino de una transferencia (parámetro *targetAccount)* no se ha establecido.

En la Tabla 1 se muestran tres posibles casos de prueba para esta clase: el del lado izquierdo, arriba (*testNormal_700)* ejercita un escenario "normal", en el que no se espera que se lance ninguna excepción; en el de la izquierda, arriba (*testNoBalanceTransfer)*, se intenta transferir más dinero del que se dispone en la cuenta; en el tercero (*testNegativeTransfer)*, se intenta transferir una cantidad negativa:

1. En el primer caso, el oráculo consta de una instrucción *assertTrue* (que comprueba si el saldo obtenido es igual al esperado, emitiendo el veredicto de paso o de fallo que corresponda) y de una instrucción *fail* (que, en caso de alcanzarse, emite veredicto de fallo).
2. En el segundo y tercer casos, y puesto que lo que se espera es que respectivamente se lance una *InsufficientBalanceException* o una *NegativeAmountException,* el oráculo consta de dos instrucciones *fail* (una en el bloque *try,* que emite veredicto de fallo si no se lanza excepción; otra en el segundo bloque *catch,* que emite veredicto de fallo si la excepción que se lanza no es la esperada) y un bloque *catch* vacío (si se captura la excepción que se esperaba, caso en el que el SUT se ha comportado

correctamente y se emite veredicto de paso).

Tabla 1. Tres casos de prueba para la clase *Account*

```
@Test
public void testNormal_700() {
  Account c=new Account();
  try {
    c.deposit(1000);
    c.withdraw(200);
    c.transfer(new Account(),
         "Telephone", 100);
    assertTrue(c.getBalance()==700);
  }
  catch (Exception e) {
    fail("No exception expected");
  }
}
@Test
public void testNoBalanceTransfer() {
  Account c=new Account();
  try {
   c.deposit(1000);
   c.withdraw(200);
   c.transfer(new Account(),
          " Telephone ", 5000);
   fail("InsufficientBalanceException "
          + "expected");
  }
  catch (InsufficientBalanceException e)
  {}
  catch (Exception e) {
    fail("InsufficientBalanceException "
          + expected");
  }
}
```

```
@Test
public void testNegativeTransfer() {
  Account c=new Account();
  try {
    c.deposit(1000);
    c.withdraw(200);
    c.transfer(new Account(),
          " Telephone ", -100);
    fail("NegativeAmountException " +
                expected");
  }
  catch (NegativeAmountException e) {}
  catch (Exception e) {
    fail("NegativeAmountException " +
                expected");
  }
}
```

Como se ve, los tres casos de prueba tienen prácticamente la misma estructura: se construye la instancia y se llama a *deposit, withdraw* y *transfer*. Pero, sin embargo, el oráculo es muy distinto en los tres casos.

En un generador automático de datos y casos de prueba, el tester puede asignar diferentes valores a los parámetros de las operaciones involucradas en los casos. Por ejemplo, si se asignan los valores {1000, 500} a *deposit,* {200, 50} a *withdraw* y {5000, 100, -100} al tercer parámetro de *transfer,* aplicando un algoritmo de generación de todas las combinaciones se podrían obtener los 2x2x3=12 casos de prueba de la Tabla 2. Entre ellos, se encuentran los tres que hemos especificado arriba.

Tabla 2. Posibles combinaciones

#	*deposit*	*withdraw*	*transfer*	#	*deposit*	*withdraw*	*transfer*
	{1000, 500}	*{200, 50}*	*{5000, 100, -100}*		*{1000, 500}*	*{200, 50}*	*{5000, 100, -100}*
1	1000	200	5000	7	500	200	5000
2	1000	200	100	8	500	200	100
3	1000	200	-100	9	500	200	-100
4	1000	50	5000	10	500	50	5000
5	1000	50	100	11	500	50	100
6	1000	50	-100	12	500	50	-100

Una vez generadas estas combinaciones y traducidas a casos de prueba, es el tester

quien debe decidir, para cada una, cuál es el oráculo que le corresponde. En la Tabla 2 mostramos tres parámetros numéricos, con lo que la determinación del oráculo es casi trivial. En situaciones reales, la variedad de escenarios, operaciones y parámetros hace ya compleja la asignación manual del oráculo, y mucho más difícil aún su determinación e inclusión automáticas.

Por otro lado, y aunque la cobertura que alcanza un test suite obtenido generando todas las combinaciones es la máxima que, con esos datos de prueba, se puede obtener, aplicar todas las combinaciones produce una cantidad enorme de casos de prueba, muchos de los cuales serán además redundantes (es decir, ejercitan exactamente las mismas áreas del código) [4]: los casos números 2, 5, 8 y 11 de la Tabla 2 corresponden a un escenario "normal", en el que se retiran y transfieren importes teniendo para ambas operaciones saldo suficiente: probablemente con utilizar uno de esos cuatro casos sería suficiente.

Por estos motivos se utilizan otras estrategias para generar casos de prueba, como algoritmos de *pairwise* [5, 6], que producen menos casos de prueba que *all combinations* pero que, si los datos de prueba están bien elegidos, alcanzan por lo general una cobertura bastante alta.

El problema viene cuando esta cobertura "bastante alta" no alcanza el umbral deseado: de acuerdo con el proceso descrito en la Figura 1, se deben crear nuevos casos de prueba que recorran las zonas inexploradas del SUT, y a estos casos de prueba hay que dotarlos de un oráculo adecuado. Este es precisamente el problema cuya solución abordamos en este trabajo.

3 Trabajos relacionados

3.1 Generación de casos de prueba

La generación automatizada de casos de prueba es una línea de investigación prolífica en ingeniería de software: como ejemplo, una búsqueda simple en Scopus para publicaciones con los términos "generación de casos de prueba" en el título arroja 170 documentos de conferencia y 61 artículos, solo de 2011 a octubre de 2015. Nueve reconocidos especialistas, dirigidos por Bertolino, Li y Zhu, han coordinado el artículo *An orchestrated survey on automated software test case generation* [7], que clasifica las técnicas para la generación de casos de prueba en cinco categorías: (1) ejecución simbólica y cobertura estructural del programa ; (2) generación basada en modelos (MBT); (3) pruebas combinatorias; (4) prueba aleatoria adaptativa (ART); y (5) pruebas de software basadas en búsqueda (SBST). A continuación se describen algunas conclusiones de esta survey:

- En cuanto a la ejecución simbólica, Anand y Harrold destacan algunos problemas importantes de esta categoría: un número extremadamente grande de caminos en el software del mundo real, variedad de lenguajes de programación, imposibilidad o dificultad para recuperar la estructura de los módulos binarios e indecibilidad de las restricciones de ruta. En sus conclusiones, los autores explican que "se necesita más investigación para mejorar la utilidad de la técnica en los programas del mundo real".

Desafortunadamente, la mayoría de los problemas de ejecución simbólica aplicados a testing están abiertos desde hace muchos años.

- En la citada survey, Grieskamp revisa las técnicas de generación de MBT. Los desafíos pendientes de esta categoría se resumen en un informe externo, escrito por el propio Grieskamp y colaboradores [8]. Los autores destacan mejoras significativas desde 2004 (año del informe anterior) hasta 2011, sobre todo en la disponibilidad de herramientas para MBT. Sin embargo, varios de los desafíos de investigación de 2004 siguen abiertos; en particular: el tratamiento del no determinismo en los modelos; la brecha entre el nivel de abstracción de los modelos y el código; repetibilidad de casos de prueba (sobre todo en entornos distribuidos y en la nube); dificultades para la trazabilidad desde los requisitos hasta los modelos; y la falta de uso de ingeniería dirigida por modelos en la industria.
- Cohen está a cargo de la revisión de los métodos de prueba combinatoria. Esta autora resalta que ésta es "una prometedora área de investigación [...] con oportunidades para mejorar nuevos dominios de aplicación". Algunas direcciones futuras fructíferas en el campo de las pruebas combinatorias incluyen "la extracción automática del modelo, la adaptación a la evolución del modelo y el desarrollo de técnicas que reutilicen o compartan información entre diferentes pruebas".
- La prueba aleatoria adaptativa (ARV) es una mejora de las pruebas aleatorias de cuya revisión, en la citada survey, se encarga Chen. La efectividad de ART con respecto a las pruebas aleatorias es significativamente mayor, pero presenta problemas de eficiencia con respecto al tiempo y los recursos computacionales. Por lo tanto, la investigación en esta línea debe centrarse en la mejora de la eficiencia de los algoritmos.
- Harman, McMinn, Clark y Burke redactan la sección sobre SBST, que se considera una rama de la ingeniería de software basada en la búsqueda. SBST utiliza algoritmos de optimización para automatizar la búsqueda de datos de prueba que maximiza el logro de los objetivos de prueba, mientras que minimiza sus costos. Según estos autores, "el principal objetivo [de estas técnicas] es la definición de una función objetivo que capture adecuadamente los requisitos de la prueba y que guíe correctamente al algoritmo de optimización basado en la búsqueda". Los algoritmos genéticos o de colonias de hormigas son técnicas que entran en esta categoría. La posibilidad de modelar con funciones de acondicionamiento físico cualquier requisito de prueba da a las técnicas de SBST "muchos objetivos de prueba emocionantes, importantes y productivos que aún no se han atacado [...], proporcionando así muchas vías fructíferas para el trabajo futuro".

3.2 El problema del oráculo

En una survey reciente, Barr et al. revisan casi cuarenta años de investigación sobre el problema del oráculo [9]. Clasifican las técnicas propuestas en 694 publicaciones de 1978 a 2012 en cuatro categorías: (1) oráculos de prueba especificados, que requieren algún tipo de especificación formal; (2) oráculos de prueba derivados, que involucran artefactos de los cuales se puede derivar un oráculo de prueba (como una versión anterior del sistema, por ejemplo); (3) oráculos implícitos, que están relacionados con la

detección de errores obvios (bloqueos del programa, excepciones de puntero nulo, etc.); y (4) falta de oráculo de prueba automatizado (oráculo humano, por ejemplo).
Según estos autores, el oráculo es un "problema abierto actual que representa un importante cuello de botella que requiere una mayor automatización de las pruebas y la adopción de métodos y herramientas de pruebas automatizadas de manera más amplia". De hecho, así como la generación de casos de prueba puede ser un problema relativamente resuelto, la adición de oráculos a cada caso de prueba permanece prácticamente sin resolver.
Bertolino [10] también enfatizó la automatización de la generación de Oracle como uno de los principales desafíos de la investigación de pruebas de software. Para ella, un "oráculo ideal" debería proporcionar "los resultados esperados para cada caso de prueba dado". Tal oráculo ideal sería un "motor o heurística que emita un veredicto de paso/fallo sobre las salidas de prueba observadas". Baresi y Young [11] también mencionaron el concepto de "oráculo ideal", que "satisfaría las propiedades deseables de las especificaciones del programa, como ser completo, evitando la sobreespecificación y permitiendo al mismo tiempo la eficiencia en las comprobaciones". Los métodos formales o los modelos completos del sistema son actualmente el método más adecuado para el desarrollo automático de oráculos, pero necesitan esa sobreespecificación mencionada por Baresi y Young. Pérez et al., por ejemplo, obtienen casos de prueba extraídos de modelos de secuencia UML en líneas de productos de software [12]. Su enfoque requiere, de hecho, una especificación bastante compleja de los escenarios de prueba con sus pre y poscondiciones, mensajes, parámetros, etc. La propuesta es económicamente viable porque es altamente reutilizable en el contexto de SPL, pero es demasiado costosa para el desarrollo de desarrollos aislados.
Pero, como Barr et al. señalan, "en la industrial, para probar la mayoría de los sistemas, el tester no dispone de las especificaciones formales, aserciones u oráculos parciales. [...] El tester se enfrenta a la abrumadora tarea de verificar manualmente el comportamiento del sistema para cada uno de los casos de prueba ".
El problema oráculo sigue siendo un importante desafío (quizá el que más) para la generación automatizada de casos de prueba. Con respecto a este problema, Annand et al. afirman que "la mayoría de los trabajos sobre automatización del testing se centran en la búsqueda de buenas entradas de prueba, pero no abordan el problema igualmente importante de reducir el costo de verificar el producto producido en respuesta a esas entradas" [7].

4 Generación de casos de prueba con oráculo

El proceso que aquí presentamos está soportado por SMACTesting, una herramienta web específicamente diseñada para la automatización de pruebas de sistemas *Social*, *Mobile*, *Analytics*, *Internet of Things* y *GIS* que se encuentren implementados en Java. Inicialmente, el tester crea en la herramienta un proyecto en el cual carga el código del SUT, incluyendo el código compilado.

Mediante un motor que, respectivamente, hace uso de la *Reflection API* de Java y de la librería Jsoup [13], la herramienta muestra al tester el conjunto de ficheros *.class* y

de páginas *html* y *jsp* contenidas en el proyecto, así como las operaciones (en los *.class)* y controles (botones, cajas de texto, etcétera) de cada uno. El tester puede seleccionar las operaciones que desee y, con ellas, escribir una expresión regular que describe un conjunto genérico de escenarios de prueba. Volviendo al ejemplo de la clase *Account,* una expresión regular como *new Account·deposit·[deposit|withdraw|transfer]** puede ser expandida para obtener, entre otras, las siguientes secuencias de operaciones (en las que, por brevedad, obviamos los dos primeros parámetros de *transfer*):

1. *a=new Account(); a.deposit(amount);*
2. *a=new Account(); a.deposit(amount); a.deposit(amount);*
3. *a=new Account(); a.deposit(amount); a.withdraw(amount);*
4. *a=new Account(); a.deposit(amount); a.transfer(amount);*
5. *a=new Account(); a.deposit(amount); a.withdraw(amount); a.transfer(amount);*

Los tres casos de prueba que pusimos como ejemplo en la Tabla 1 proceden de la secuencia número 5 pero, obviamente, los parámetros de las operaciones involucradas deben anotarse con valores de prueba para obtener casos efectivamente ejecutables. De este modo, con los valores que dábamos como ejemplo en la Tabla 2 para generar todas las combinaciones, de la secuencia número uno obtendríamos 2 casos de prueba, 4 de la segunda y tercera, 6 de la cuarta y los 12 que mostrábamos en la Tabla 2 en el caso de la quinta. SMACTesting incluye el mecanismo necesario para asignar los valores a los parámetros de las operaciones seleccionadas.

4.1 Creación de oráculos y plantillas de prueba

Un caso de prueba debe comprobar (1) que el sistema hace lo que debe hacer y (2) que no hace lo que no debe hacer. En general:

- Los casos del primer tipo incluyen un bloque *try* en el que se ejercita el escenario y se realiza una comprobación (mediante un *assert).* Después se incluye un *catch* en el que se emite con *fail* un veredicto de fallo, pues si se ha llegado a este *catch* es porque se ha lanzado una excepción que no se esperaba. Así pues, los casos de prueba de este estilo siguen el modelo del caso *testNormal_700* de la Tabla 1.
  En efecto, la plantilla básica con la que SMACTesting genera estos casos es la de la Figura 2. El token *TCNUMBER* se sustituirá más adelante por un número de caso de prueba. El token <*sequence/*> se sustituye por la secuencia de operaciones que procedan de la secuencia que se esté tratando.

```
@Test
public void testDefaultTCNUMBER() {
  try {
    <sequence/>
  }
  catch (Exception e) {
    fail("No exception expected");
  }
}
```

Fig. 2. Plantilla *testDefault,* para casos que prueban que *el sistema hace lo que debe hacer*

- Los casos del segundo tipo incluyen un bloque *try* con un *fail* al final (pues si se ha llegado a ese punto es porque no se ha lanzado una excepción que se esperaba), un *catch* que recoge la excepción esperada (y que emite veredicto de paso) y otro *catch* que captura cualquier otro tipo de excepción (y que lanza veredicto de fallo, pues la excepción es diferente de la esperada). De este modo, los casos de prueba de esta categoría siguen el modelo de los casos *testNoBalanceTransfer* y *testNegativeTransfer* de la Tabla 1.

  La plantilla básica para estos casos es la de la Figura 3. Igual que antes, los tokens *TCNUMBER y <sequence/>* se sustituirán más tarde por el número de caso de prueba que corresponda y por la secuencia de operaciones que procedan de la secuencia que se esté tratando. El token *EXPECTED_EXCEPTION* se sustituye por la excepción esperada.

```
@Test
public void testException TCNUMBER() {
    try {
    <sequence/>
    fail("EXPECTED_EXCEPTION expected");
  }
  catch (EXPECTED_EXCEPTION e) {}
  catch (Exception e) {
    fail("EXPECTED_EXCEPTION expected");
  }
}
```

Fig. 3. Plantilla *testException,* para casos que prueban que *el sistema no hace lo que no debe hacer*

Una vez que el tester ha seleccionado las operaciones, asignado valores a sus parámetros y escrito la expresión regular, un motor de generación expande la expresión regular obteniendo las secuencias de operaciones. A continuación, el tester debe describir oráculos genéricos que asignará a la expresión regular. Cuando, más adelante, se generen los casos de prueba, SMACTesting irá comprobando, para cada uno, qué oráculo le corresponde, y generará su código conforme a la plantilla.

Los oráculos se describen mediante *cláusulas when,* que determinan, en términos de los valores de los parámetros, con qué plantilla de caso de prueba (figuras 2 y 3) debe generarse el código.

Así, una cláusula *when* para el ejemplo de la cuenta corriente y los valores de la Tabla 2 es el siguiente:

when transfer.amount = 5000, usar plantilla testException, siendo EXPECTED_EXCEPTION = InsufficientBalanceException

Lo que este *when* especifica es que cuando valor del parámetro *amount* de la operación *transfer* sea 5000 (que es uno de los valores de prueba que el tester seleccionó para este parámetro), el caso de prueba debe generarse usando la plantilla *testException,* y sustituyendo el token *EXPECTED_EXCEPTION* por *InsufficientBalanceException.*

En SMACTesting pueden definirse tres tipos de cláusulas *when.* Si bien en la herra-

mienta se describen visualmente mediante su interfaz gráfica, los presentamos a continuación de manera textual:

1. Los *when* de tipo *OR* se verifican cuando cualquiera de las condiciones que contienen es cierta. Suponiendo que hubiéramos asignado el valor -100 a los tres parámetros numéricos de las tres operaciones, podríamos especificar lo siguiente:

```
when deposit.amount = -100 OR
        withdraw.amount = -100 OR
        transfer.amount = -100,
        usar plantilla testException, siendo
                EXPECTED_EXCEPTION = NegativeAmountException
```

2. Los *when* de tipo *AND* se verifican cuando todas las condiciones incluidas en la cláusula son ciertas. La siguiente cláusula *when* indica que el caso de prueba en el que se ingresan 1000, se retiran 200 y se transfieren 100 debe generarse con la plantilla *testNormal.*

```
when deposit.amount = 1000 AND  withdraw.amount = 200
AND transfer.amount = 100,  usar plantilla testDefault
```

   Tanto las cláusulas *OR* como las *AND* admiten paréntesis. Así, suponiendo que los valores utilizados hubieran sido asignados a los parámetros, la siguiente cláusula indica que los caos en los que se ingresen 1000 o 2000, se retiren 200 o 100, y se transfieran 100, deben generarse con la plantilla *testNormal.*:

```
when (deposit.amount = 1000 OR deposit.amount = 2000) AND
        (withdraw.amount = 200 OR withdraw.amount = 100) AND
        transfer.amount = 100,
        usar plantilla testDefault
```

3. Los *when* de tipo *ELSE* se utilizan para generar el código de todos los casos de prueba que no correspondan con ninguna de las cláusula *when* de tipo *OR* o *AND* que se hayan descrito. La plantilla para estos casos de prueba (la básica se muestra en la Figura 4) emite siempre un veredicto de fallo, pues ha ejercitado un escenario del SUT que el tester no había previsto.

```
@Test
public void testElseTCNUMBER() {
    try {
        <sequence/>
        fail("Unexpected situation");
    }
    catch (Exception e) {
        fail("Unexpected situation");
    }
}
```

Fig. 4. Plantilla *testElse,* para casos que prueban que ejercitan escenarios imprevistos

4.2 Algoritmo genético

Cuando el motor de combinación ha generado los casos de prueba, cada uno a partir de la plantilla de caso de prueba que corresponda según su cláusula *when,* el tester puede pasar a ejecutar los casos. Si opta por ejecutarlos con un algoritmo genético, el motor de ejecución irá pasando los resultados a un motor de enriquecimiento, que toma los casos de prueba originales, los combina genéticamente y produce una nueva población (es decir, un nuevo test suite), que envía al motor de ejecución. El proceso se detiene cuando se alcanza el mutation score que se haya preestablecido o cuando se haya alcanzado el número de iteraciones que el tester haya prefijado. Es importante notar que los operadores de mutación se introducen mediante programación orientada a aspectos, evolucionado a partir del que ya presentamos hace unos años en este mismo foro [14].

Codificación de individuos. Siendo el test suite la población del algoritmo genético, cada caso de prueba es un individuo de esa población. Desde el punto de vista de su codificación genética, el caso de prueba tiene su secuencia de operaciones y, para cada una, los valores de sus parámetros. Además, cada parámetro mantiene un puntero a los valores que el tester estableció para ese parámetro. La Figura 5 ilustra dos casos de prueba codificados: el primero corresponde a una situación normal (y su código se habrá generado con la plantilla *testDefault)*, y el segundo a una situación de excepción.

Valores de prueba que estableció el tester		
deposit: {1000, 500, 100}	**withdraw:** {200, 50}	**transfer:** {5000, 100, -100}

testDefault1				
Operación	*new Account*	*deposit*	*withdraw*	*transfer*
Valor		*1000*	*200*	*100*

testException2				
Operación	*new Account*	*deposit*	*withdraw*	*transfer*
Valor		*500*	*50*	*-100*

Fig. 5. Ejemplo de codificación de dos individuos

Función de cruce. La función de cruce produce dos casos de prueba hijos a partir de dos progenitores. Se genera un punto de corte aleatorio por el que se parten los dos individuos. El primer hijo se forma con la porción izquierda (respecto del punto de corte) del primer progenitor y la porción derecha del segundo. El segundo hijo se forma con la porción izquierda del segundo progenitor y la porción derecha del segundo. Suponiendo que, siendo los progenitores los dos casos de prueba de la Figura 5, y asumiendo que el punto de corte aleatorio está justo en la mitad, la Figura 6 muestra los dos descendientes que se obtendrían.

testHijo1_1				
Operación	*new Account*	*deposit*	*withdraw*	*transfer*
Valor		*1000*	*50*	*-100*

testHijo2_1				
Operación	*new Account*	*deposit*	*withdraw*	*transfer*
Valor		*500*	*200*	*100*

Fig. 6. Cruce de los dos individuos de la Figura 6

Función de mutación genética. Los nuevos descendientes se van mutando con una pequeña probabilidad. Si a un individuo le toca mutarse, se selecciona al azar uno cualquiera de sus parámetros, y se sustituye por otro valor al azar de entre los que suministró el tester. En la Figura 7, el valor inicial pasado a *deposit* (1000) se ha sustituido por 100.

Valores de prueba que estableció el tester		
deposit: {1000, 500, 100}	**withdraw:** {200, 50}	**transfer:** {5000, 100, -100}

testHijo1_1				
Operación	*new Account*	*deposit*	*withdraw*	*transfer*
Valor		*1000*	*50*	*-100*

testHijo1_1 (mutado)				
Operación	*new Account*	*deposit*	*withdraw*	*transfer*
Valor		*100*	*200*	*100*

Fig. 7. Mutación del primer hijo

Función de fitness. La función de fitness se calcula en función del número de mutantes que mata el caso de prueba. Como se comenta en la sección 5 (conclusiones y trabajo futuro), es un elemento de estudio y *tunning* muy interesante.

Algoritmos de cruce. Se han implementado tres algoritmos de cruce en SMACTesting:

1. *BestToWorst:* partiendo de los casos de prueba ordenados por fitness, elige como pareja de progenitores al primer individuo (el mejor) y al último (el peor); luego, al segundo con el penúltimo; al tercero con el antepenúltimo, etcétera.
2. *BestWithBest:* partiendo de los casos de prueba ordenados por fitness, cruza al primero con el segundo, al tercero con el cuarto, etcétera.
3. *RandomCrosser:* cruza a los individuos aleatoriamente.

Proceso genético. Independientemente del algoritmo de cruce seleccionado, el algoritmo genético procede siempre de la misma manera (Figura 9): se comienza ejecutando el test suite y construyendo la matriz de muertos (*killingMatrix)*. Si no se alcanza la condición de parada, la matriz se ordena por *fitness* y se selecciona un porcentaje de los mejores individuos (*elite)*, que son los que se cruzan para pasar a la nueva población. Estos individuos se mutan genéticamente conforme al procedimiento descrito con una probabilidad baja (en la figura, el 5%). Cuando un individuo es mutado, es necesario proporcionarle una plantilla para generar su código, que se le busca y asigna mediante

la función *buscarPlantillaPara.* La nueva población se construye con la élite de la generación anterior y los descendientes de ésta, contenidos en la *nuevaPoblación.*

```
Sean: TS el test suite, t un caso de prueba, template una plantilla de caso
de prueba

iteraciones=1;
do {
    killingMatrix = execute(TS)
    if (mutationScore(killingMatrix)>=umbral || iteraciones>I)
       break;
    ordenarPorFitness(killingMatrix)
    elite = seleccionarElite(%)
    nuevaPoblación = cruzar(elite)
    for i=0 to |nuevaPoblación|
       t = nuevaPoblación [i]
       if random()<0.05 {
          mutate(t)
          template = buscarPlantillaPara(t)
       }
    }
    TS = elite ∪ nuevaPoblación
    iteraciones = iteraciones + 1
} while (true)
```

Fig. 8. Pseudocódigo del algoritmo genético

Como puede apreciarse, la aplicación de este algoritmo nunca generará casos de prueba que no hubieran podido ser generados mediante todas las combinaciones. No obstante, en los primeros experimentos hemos comprobado que, incluso a partir de un test suite inicial muy pequeño (generado incluso con el algoritmo de combinación *Each choice* [4], el tiempo total de ejecución para conseguir un test suite que alcance el umbral de mutación preestablecido es mucho menor que ejecutando absolutamente todos los casos de prueba posibles.

5 Conclusiones y trabajo futuro

En este artículo se ha descrito un algoritmo genético que, junto a otras técnicas de prueba, se encuentra implementado en SMACTesting, una herramienta que se encuentra en estos momentos en su última etapa de desarrollo, casi lista para pasar a producción. Es por ello por lo que aún no podemos ofrecer datos concretos de validación experimental, si bien los primeros resultados, obtenidos informalmente, son muy prometedores.

Con respecto del algoritmo genético, en la próxima fase experimental deseamos:

1. Estudiar diferentes métodos de cálculo del *fitness*: con el criterio actual, no se seleccionan los casos de prueba que, aún matando pocos mutantes, matan a mutantes que no matan a otros. Así, por ejemplo, sería conveniente pasar a la siguiente generación a un caso de prueba que es el único que mata un determinado mutante.
2. Estudiar el porcentaje de elite (mejores individuos), así como la conveniencia o no de pasar a toda la elite a la siguiente generación.
3. Como se ha comentado, nuestro algoritmo genético nunca producirá un test suite que alcance un mutation score mayor que el obtenido con todas las combinaciones. çno

obstante, como el tiempo de ejecución sí es menor, es necesario extender la potencia de las cláusulas *when* permitiendo la descripción de más situaciones con operadores como <, >, <=, <>, etcétera, de manera que la mutación genética pueda insertar valores diferentes del conjunto de valores iniciales.

Agradecimientos

Este trabajo es parte de los proyectos GINSENG (TIN2015-70259-C2-1) y SMACTesting (programa Innterconecta, EXP-00082064 / ITC-20151305), financiados por el Ministerio de Economía, Industria y Competitividad y fondos FEDER.

Referencias

1. IEEE Computer Society: IEEE Standard 829 for Software and System Test Documentation. (2008).
2. Beck, K., Gamma, E.: Test-infected: programmers love writing tests. In: More Java Gems. Cambridge University Press (2000).
3. Polo, M., Reales, P.: Mutation Testing Cost Reduction Techniques: A Survey. IEEE Softw. 27, 80–86 (2010).
4. Grindal, M., Offutt, J., Andler, S.F.: Combination testing strategies: A survey. Software Testing, Verification, and Reliability. 15, 167–199 (2005).
5. Cohen, D., Society, I.C., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG System: An Approach to Testing Based on Combinatorial Design. IEEE Transactions on Software Engineering. 23, 437–444 (1997).
6. Pérez Lamancha, B., Polo, M., Piattini, M.: PROW: A Pairwise algorithm with constRaints, Order and Weight. Journal of Systems and Software.
7. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., Bertolino, A., Jenny Li, J., Zhu, H.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software. 86, 1978–2001 (2013).
8. Grieskamp, W., Hierons, R.M., Pretschner, A.: 10421 Summary -- Model-Based Testing in Practice. Dagstuhl Seminar Proceedings. (2011).
9. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering. 41, 507–525 (2015).
10. Bertolino, A.: Software Testing Research: Achievements, Challenges, Dreams. In: Future of Software Engineering, 2007. FOSE '07. pp. 85–103 (2007).
11. Luciano Baresi, Michal Young: Test oracles, Technical Report CIS-TR-01-02, http://ix.cs.uoregon.edu/~michal/pubs/oracles.pdf, (2001).
12. Pérez Lamancha, B., Polo, M., Caivano, D., Piattini, M., Visaggio, G.: Automated generation of test oracles using a model-driven approach. Information and Software Technology. 55, 301–319 (2013).
13. jsoup Java HTML Parser, with best of DOM, CSS, and jquery, https://jsoup.org/.
14. Polo Usaola, M.: Using Aspect-Oriented Programming for mutation testing of third party components. In: Proc. of the 17th CIBSE. pp. 247–261. , Pucón, Chile (2014).