

Aserciones de tests vs contratos en el testing de regresión: Un estudio de efectividad

Claudio Dosantos

Departamento de Computación
Facultad de Cs. Exactas Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Director: Dr. Facundo Molina
Codirector: Dr. Nazareno Aguirre

10 de septiembre de 2023

Capítulo 1

Introducción

En esta sección, se presenta una introducción al problema del oráculo, los diferentes tipos de oráculos y cómo se mide la calidad de los mismos.

1.1. Problema del oráculo

El problema del oráculo es...

Referencia: [?]

1.2. Tipos de Oráculos

Existen varios tipos de oráculos, tales como...

1.3. Calidad de los Oráculos

La calidad de los oráculos se mide mediante...

Capítulo 2

Generación Automática de Oráculos

La generación automática de oráculos es una técnica innovadora en el campo de la verificación y validación de software que busca abordar uno de los desafíos más complejos en el proceso de pruebas: la creación confiable y precisa de oráculos de prueba.

En el contexto de las pruebas de software, un oráculo es un mecanismo que compara los resultados esperados de un programa con los resultados reales obtenidos durante la ejecución de las pruebas. El oráculo determina si el programa se comporta correctamente o si existen discrepancias que indiquen la presencia de errores o fallos.

La generación automática de oráculos tiene como objetivo reducir la intervención manual en la creación de estos oráculos, lo que a menudo es una tarea laboriosa y propensa a errores. Utilizando un conjunto diverso de técnicas que incluyen inteligencia artificial, machine learning, análisis estático, dinámico y también técnicas aleatorias, esta metodología permite inferir o construir automáticamente oráculos precisos y efectivos para diferentes tipos de pruebas.

2.1. Unit Assertions con Randoop/EvoSuite

2.1.1. Randoop

Randoop propone una mejora a la generación de pruebas aleatorias mediante la utilización de una técnica llamada generación de pruebas aleatorias dirigidas por retroalimentación (Feedback-directed Random Test Generation).

El enfoque de Randoop se basa en generar secuencias de invocaciones a métodos de las clases bajo prueba de forma aleatoria. Estas secuencias, que se crean inicialmente, luego se combinan para obtener nuevas secuencias. A continuación, se someten a una serie de evaluaciones mediante contratos y filtros, como la detección de excepciones, errores de aserciones o incluso la comparación con secuencias previamente generadas. Las secuencias que superan estas validaciones son seleccionadas como candidatas prometedoras para generar nuevas secuencias y, posteriormente, se pueden utilizar como tests de regresión para asegurar la calidad y funcionalidad continua del sistema bajo prueba.

Por otro lado, las secuencias que no cumplen con uno o más de los contratos establecidos indican la presencia potencial de errores en el sistema bajo prueba. Estas secuencias problemáticas se convierten en valiosos hallazgos para los equipos de desarrollo, ya que apuntan a áreas críticas que requieren correcciones y mejoras.

El enfoque de generación de pruebas aleatorias dirigidas por retroalimentación de Randoop representa un intento significativo para mejorar la efectividad del testing aleatorio al combinarlo con criterios de selección más inteligentes y evaluaciones de calidad. Al hacerlo, se logra una mayor cobertura de pruebas y se descubren más escenarios de prueba, lo que puede llevar a una mejor detección y corrección de errores en el software bajo evaluación. Este enfoque se ha mostrado prometedor en muchos entornos de desarrollo y se considera una herramienta valiosa para la mejora del proceso de testing en el desarrollo de software.

2.1.2. EvoSuite

EvoSuite es una herramienta de generación automática de conjuntos de pruebas para software orientado a objetos, que emplea un enfoque basado en algoritmos de búsqueda. Esta técnica combina diversas estrategias, como búsqueda híbrida, ejecución simbólica dinámica y transformación de testabilidad, para lograr una generación de pruebas más eficiente y efectiva.

Uno de los aspectos clave de EvoSuite es la utilización de algoritmos genéticos en su proceso de generación de pruebas. Estos algoritmos genéticos se inspiran en la evolución biológica y aplican operadores genéticos como selección, cruce y mutación en poblaciones de pruebas para mejorar la cobertura y calidad de los casos de prueba generados.

EvoSuite emplea dos técnicas principales:

1. Generación del conjunto de pruebas completo (Whole test suite generation): En lugar de optimizar metas de cobertura individuales, EvoSuite se enfoca en optimizar un criterio de cobertura general. Esto evita que el resultado se vea negativamente influenciado por el orden o la dificultad de alcanzar metas de cobertura individuales. Al optimizar la cobertura de manera integral, la herramienta puede producir conjuntos de pruebas más completos y representativos.

2. Generación de aserciones basada en mutaciones (Mutation-based assertion generation): Esta técnica se centra en evaluar la efectividad de los casos de prueba generados. Para ello, se ejecutan las pruebas en unidades de prueba (UUT) con algunas modificaciones o mutaciones en el código. Aquellas pruebas que detecten los errores introducidos por las mutaciones reciben una mejor puntuación, ya que se consideran más valiosas. Finalmente, se selecciona el subconjunto más pequeño de pruebas que es capaz de detectar las mutaciones, lo que garantiza una cobertura efectiva y eficiente de las posibles fallas en el software.

La combinación de búsqueda híbrida, ejecución simbólica dinámica y transformación de testabilidad, junto con el uso de algoritmos genéticos, hacen que EvoSuite sea una herramienta poderosa para la generación automática de conjuntos de pruebas. Esta aproximación ha demostrado ser efectiva en diversos contextos de desarrollo de software, y ha ayudado a mejorar la calidad y confiabilidad de los programas bajo prueba.

2.2. Contratos con Daikon

Daikon es una herramienta que emplea técnicas de aprendizaje automático para llevar a cabo la detección dinámica de posibles invariantes en el código de programas. Utiliza datos recopilados durante múltiples ejecuciones del programa para analizar patrones y descubrir propiedades que se mantienen constantes a lo largo de todas las ejecuciones.

A través de un proceso de aprendizaje automático, Daikon busca identificar invariantes relevantes que puedan ser representativos del comportamiento general del programa. Estas propiedades se consideran como reglas generales que deben cumplirse en todas las instancias del programa.

Daikon utiliza una gramática de propiedades y variables para definir qué tipo de invariantes buscar. Las propiedades pueden ser de diferentes formas, como comparaciones numéricas, relaciones entre variables, o incluso estructuras de datos complejas. Las variables representan los valores que se recopilan durante la ejecución del programa.

Las aplicaciones de Daikon son diversas y abarcan desde la documentación del programa, hasta la detección y prevención de errores (bugs), debugging, testing, verificación de propiedades, análisis de estructuras de datos y corrección de inconsistencias.

2.3. Contratos con SpecFuzzer

SpecFuzzer es una técnica que se utiliza para...

2.4. Ejemplos de generación de oráculos

```

1  /**
2   * Array-based implementation of the stack.
3   * @author Mark Allen Weiss
4   */
5  public class StackAr {
6      /**
7       * Construct the stack.
8       * @param capacity the capacity.
9       */
10     public StackAr( int capacity )
11     {
12         theArray = new Object[ capacity ];
13         topOfStack = -1;
14     }
15
16     /**
17      * Test if the stack is logically empty.
18      * @return true if empty, false otherwise.
19      * @observer // annotation added by Jeremy
20      */
21     public boolean isEmpty( )
22     {
23         return topOfStack == -1;
24     }
25
26     /**
27      * Get the most recently inserted item in the stack.
28      * Does not alter the stack.
29      * @return the most recently inserted item in the stack, or null, if
30      *         empty.
31      * @observer // annotation added by Jeremy
32      */
33     public Object top( )
34     {
35         if( isEmpty( ) )
36             return null;
37         Object result = theArray[ topOfStack ];
38         assert(true);
39         return result;
40     }
41 }

```

Código 2.1: Ejemplo StackAr.top

2.4.1. Randoop

```

1  @Test
2  public void test062() throws Throwable {
3      if (debug)
4          System.out.format("%n%s%n", "RegressionTest0.test062");
5      DataStructures.StackAr stackAr1 = new DataStructures.StackAr(6);
6      boolean boolean2 = stackAr1.isEmpty();

```

```
7      stackAr1.push((java.lang.Object) 5L);
8      boolean boolean5 = stackAr1.isEmpty();
9      stackAr1.pop();
10     java.lang.Object obj7 = stackAr1.top();
11     stackAr1.makeEmpty();
12     org.junit.Assert.assertTrue("'" + boolean2 + "' != '" + true + "'",
        boolean2 == true);
13     org.junit.Assert.assertTrue("'" + boolean5 + "' != '" + false + "'",
        boolean5 == false);
14     org.junit.Assert.assertNull(obj7);
15 }
```

Código 2.2: Aserción generada con Randoop

2.4.2. Daikon

```
1 DataStructures.StackAr.top():::EXIT76
2 daikon.Quant.eltsEqual(this.theArray, null)
3 daikon.Quant.eltsEqual(daikon.Quant.typeArray(this.theArray), null)
4 this.topOfStack == -1
5 \result == null
6 daikon.Quant.eltsEqual(this.theArray, \result)
```

Código 2.3: Aserción generada con Daikon

Capítulo 3

Mutation Testing

El concepto de Pruebas de Mutación (Mutation Testing) se introdujo en la década de 1970 como un método para evaluar la efectividad de un conjunto de pruebas en la detección de fallas y para identificar sus puntos débiles.

El enfoque de las Pruebas de Mutación implica la inserción deliberada de defectos comunes en el código fuente, creando así versiones modificadas del programa llamadas mutantes. Luego, estos mutantes se someten a la ejecución del conjunto de pruebas existente, con el propósito de verificar si los defectos son detectados por las pruebas o no.

Un mutante que es detectado y produce un fallo se considera "muerto" se descarta como no relevante. Por el contrario, aquellos mutantes que no son detectados (llamados "vivos") revelan posibles debilidades en el conjunto de pruebas, ya que representan fallas potenciales no detectadas.

La calidad de un conjunto de pruebas se evalúa según la cantidad de mutantes detectados. Si un conjunto de pruebas no logra detectar mutantes, se considera débil y propenso a contener fallas no detectadas. Por el contrario, cuantos más mutantes sean detectados por las pruebas, mayor será la confiabilidad del conjunto de pruebas y la capacidad para encontrar posibles errores.

Por todas estas razones, utilizar Mutation Testing para evaluar la calidad de las pruebas y los invariantes generados es un enfoque confiable y efectivo. Al someter el software a mutantes deliberados y analizar cómo las pruebas responden ante estas versiones modificadas, se puede obtener una evaluación objetiva de la capacidad de detección de fallas del conjunto de pruebas y, en consecuencia, identificar áreas de mejora en el proceso de pruebas y desarrollo de software.

3.1. Major

Major es un framework usado para el análisis de mutación en el ámbito de la programación y prueba de software.

3.1.1. Operadores de mutación

Major incluye los siguientes operadores de mutación:

- AOR (Arithmetic Operator Replacement), el cual reemplaza operadores aritméticos binarios por otros compatibles. Ej: $a + b \Rightarrow a - b$
- COR (Conditional Operator Replacement), el cual reemplaza operadores condicionales por otros compatibles. Ej: $a \text{ --- } b \Rightarrow a \text{ } b$
- LOR (Logical Operator Replacement), el cual reemplaza operadores lógicos por otros compatibles. Ej: $a \text{ } b \Rightarrow a \& b$ *ROR (Relational Operator Replacement), el cual reemplaza operadores relacionales por otros compatibles. Ej: $a == b \Rightarrow a > b$*

- SOR (Shift Operator Replacement), el cual reemplaza operadores de desplazamientos de bits por otros compatibles. Ej: $a \ll b \Rightarrow a \ll b$
- ORU (Operator Replacement Unary), el cual reemplaza operadores unarios por otros compatibles. Ej: $-a \Rightarrow -a$
- LVR (Literal Value Replacement), el cual reemplaza un literal por un valor por defecto. Ej: $val = 0 \Rightarrow 1$ and -1
- EVR (Expression Value Replacement), el cual reemplaza una expresión por un valor por defecto. Ej: $return a \Rightarrow return 0$
- STD (Statement Deletion), Omite sentencias simples del tipo `return`, `break`, `continue`, llamadas a métodos, asignaciones, `pre` y `pos` incremento.

3.1.2. Para el ejemplo de la sección 2

```

1  31: COR: isEmpty(): TRUE: DataStructures.StackAr@top(): 75: isEmpty() |==> false
2  32: COR: isEmpty(): FALSE: DataStructures.StackAr@top(): 75: isEmpty() |==> true
3  33: STD: <RETURN>: <NO-OP>: DataStructures.StackAr@top(): 76: return null; |==>
    <NO-OP>
4  34: EVR: <ARRAY_ACCESS(java.lang.Object)>: <DEFAULT>: DataStructures.StackAr@top(): 77: theArr
    |==> null
5  35: LVR: TRUE: FALSE: DataStructures.StackAr@top(): 78: true |==> false
6  36: EVR: <IDENTIFIER(java.lang.Object)>: <DEFAULT>: DataStructures.StackAr@top(): 79: result
    |==> null

```

Código 3.1: Mutaciones StackAr.top

Capítulo 4

Análisis de Efectividad de los Oráculos

En esta sección, se presenta el diseño del experimento realizado para evaluar la efectividad de los oráculos generados automáticamente. También se presentan los resultados obtenidos.

4.1. Evaluación Experimental

Se describe el diseño del experimento y cómo se evaluaron los oráculos generados.

4.1.1. Generación de oráculos

Para comparar la eficiencia de pruebas de regresión generadas con distintos enfoques, aserciones de test y contratos se plantean los siguientes pasos.

Dado un código que representa una solución correcta a un problema específico, como primer paso se deben generar casos de pruebas utilizando las herramientas de generación automática mencionadas en el Capítulo 2.

1) Generación de tests utilizando Randoop. A partir del archivo ejecutable del código bajo prueba, la clase y el método a probar. Obtendremos aserciones como hemos visto en el Código 2.2 de la sección 2.4.1.

```
1 java -cp ${randoop_jar}:${project_classpath} randoop.main.Main gentests
   $test_classes --classlist=$classlist --serialize-folder=$mutator_inputs
   --serialize-method=\"$regex_method\" --junit-package-name=$package
   --junit-output-dir=$outdir_tests --junit-reflection-allowed=false
   --time-limit=$timelimit --literals-level=ALL --literals-file=$literals
   --omit-methods=\"$omitmethods\"
```

Código 4.1: Generación de tests con Randoop

2) Generación de aserciones utilizando Daikon. A partir de los archivos ejecutables de las pruebas generadas con Randoop, el código bajo prueba, el nombre de la clase y el método a probar, al ejecutar el siguiente script se obtienen aserciones Daikon como se ha visto en el Código 2.3 de la sección 2.4.2.

```
1 # First step: perform dynamic comparability analysis (with DynComp)
2 java -cp \
3   ${tests_bin}:${daikon_path}/daikon.jar:${subject_jar} \
4   daikon.DynComp --output-dir=${output_dir}daikon/ \
5   ${package}.RegressionTestDriver
6
```

```

7 # Second step: obtain the dtrace file with Chicory
8 echo '-->Going to run Chicory'
9 java -cp \
10 ${tests_bin}:${daikon_path}/daikon.jar:${subject_jar} \
11 daikon.Chicory --comparability-file=\
12 ${output_dir}daikon/RegressionTestDriver.decls-DynComp \
13 --output-dir=${output_dir}daikon/ ${package}.RegressionTestDriver
14
15 # Third step: actual Daikon execution to infer specs
16 echo '-->Going to run Daikon'
17 java -cp ${daikon_path}/daikon.jar daikon.Daikon \
18 --ppt-select-pattern \
19 ${package}'\.'$2'::CLASS|'${package}'\.'$2'::OBJECT|\
20 '\.'$2'\.'$method_without_args \
21 -o ${output_dir}daikon/res.inv.gz \
22 ${output_dir}daikon/RegressionTestDriver.dtrace.gz
23
24 # Process result and print the Daikon specs for better readability. First
25 # the ones related to class invariants, and then the ones related to the
   current method
26 java -cp ${daikon_path}/daikon.jar daikon.PrintInvariants \
27 ${output_dir}daikon/res.inv.gz --format java > ${output_dir}daikon/res.txt

```

Código 4.2: Generación de tests con Randoop

3) Generación y compilación de mutantes utilizando Major.

```

1 $major/bin/javac -cp $subject_jar -nowarn -J-Dmajor.export.mutants=true
   -XMutator:ALL \
2 -d ${output_dir}bin ${source_dir}src/main/java/${class_path}/${2}.java

```

Código 4.3: Generación de mutantes con Major

4) Ejecución de las aserciones generadas por Randoop sobre cada mutante. Para cada ejecución se analiza la salida, básicamente puede obtenerse como resultado que las aserciones no encuentran errores o por el contrario si lo encuentran, es decir que detectan la mutación. En este último caso, a su vez es necesario identificar aquellas ejecuciones en las que las aserciones fallan, de las que son ejecuciones erróneas. En la siguiente tabla se puede visualizar un ejemplo del resultado obtenido.

mutant id	mutation	failing test	assertions failures	non assertions failures
1	STD	300	10	290
2	LVR	26	16	10
3	LVR	26	16	10
4	STD	94	37	57
5	LVR	26	16	10
6	LVR	26	16	10
7	ROR	26	16	10
8	ROR	232	29	203
9	ROR	0	0	0
...

5) Ejecución de las aserciones generadas por Daikon sobre cada mutante. En la siguiente tabla se pueden ver los resultados obtenidos para el ejemplo.

mutant id	mutation	failing test
1	STD	6
2	LVR	0
3	LVR	0
4	STD	0
5	LVR	0
6	LVR	0
7	ROR	0
8	ROR	0
9	ROR	0
...

4.2. Resultados

Se presentan los resultados obtenidos en el experimento y se analizan.

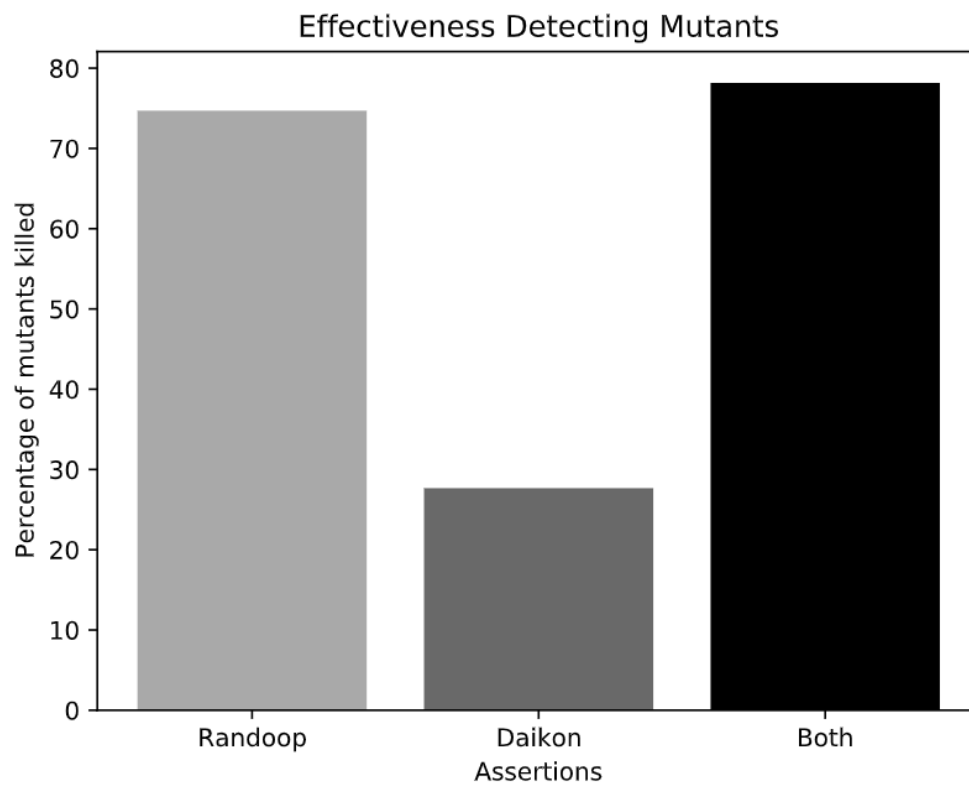


Figura 4.1: Resultados obtenidos

Herramientas	Mutantes detectados	Efectividad
Daikon	536 de 1937	27.67
Randoop	1447 de 1937	74.7
Ambas	1514 de 1937	78.16

Graficos y Tablas segun mutante

Tipo	Mutantes	Daikon	Randoop	Ambas
AOR	376	133	320	331
COR	194	48	147	159
EVR	165	38	122	126
LOR	8	2	4	4
LVR	463	129	351	364
ORU	9	4	8	8
ROR	376	78	246	258
SOR	2	2	2	2
STD	344	102	247	262

Capítulo 5

Conclusiones

En esta sección, se presentan las principales conclusiones basadas en los resultados obtenidos en el experimento. También se mencionan posibles direcciones para trabajo futuro.