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

7 de octubre de 2023

Capítulo 1

Introducción

La calidad de los sistemas de software típicamente se define alrededor de varios aspectos, tales como confiabilidad, usabilidad, eficiencia, etc. Entre ellos, la confiabilidad es generalmente considerada una característica fundamental en lo que respecta a la calidad del software, y de gran interés en el proceso de desarrollo de software [0].

Analizar la calidad del software está fuertemente relacionado con la tarea de encontrar defectos en el mismo, es decir, identificar algún comportamiento actual del software que difiera del comportamiento esperado. En este sentido, si se considera un determinado dato de entrada para un sistema, el desafío de distinguir el comportamiento correcto de uno incorrecto es conocido como el problema del oraculo []. Es esperable que esta tarea se pueda realizar con el menor costo posible y con los mayores beneficios, garantizando que el sistema funcione correctamente para distintos valores de entrada. Se han introducido técnicas para oráculos automáticos como modelado, especificaciones, desarrollo guiado por contratos, y testing metamorfico. Aún así existen casos donde ninguna de estas técnicas son adecuadas, por lo que la intervención manual del desarrollador es necesaria.

Las especificaciones de software pueden aparecer de diversas maneras. A nivel de código fuente, cuando están presentes, generalmente se manifiestan o bien como comentarios, es decir, descripciones informales en lenguaje natural de lo que se supone que debe hacer el software, o más formalmente como aserciones de programas, es decir, declaraciones (normalmente ejecutables) que capturan propiedades que el software debe satisfacer en ciertos puntos durante su ejecución. Las especificaciones como comentarios son más comunes, pero tiene la desventaja de que no pueden utilizarse de manera directa para el análisis automático de confiabilidad. Deido a esta situación, el problema de inferencia de especificaciones (un caso especial del conocido problema del oráculo [Barr et al. 2015]), es decir, el problema de generar una descripción formal del comportamiento del software a partir de elementos existentes (documentación, código fuente, etc.), ha recibido una atención cada vez mayor por parte de la comunidad de Ingeniería de Software.

En las últimas décadas, con el objetivo de ayudar a los desarrolladores a equipar las implementaciones con especificaciones, se han desarrollado una variedad de técnicas de inferencia automática de aserciones de regresión, es decir, aserciones que capturan el comportamiento actual del software. Estas técnicas, generalmente basadas en observaciones de ejecuciones del software bajo análisis, pueden producir distintos tipos de aserciones de regresión. Por ejemplo, algunas técnicas de generación automática de tests [Fraser Arcuri 2011, Pacheco Ernst 2007] incorporan mecanismos para producir aserciones de test, es decir, aserciones que capturan propiedades válidas en un escenario (test) específico. Por otro lado, otras técnicas de inferencia de especificaciones [Ernst et al. 2007, Le Lo 2018, Molina et al. 2019, Molina et al. 2021, Molina et al. 2022, Terragni et al. 2020] son capaces de inferir aserciones de contratos, es decir, aserciones más generales asociadas con elementos de contratos como precondiciones, postcondiciones e invariantes [Meyer 1992].

1.1. Problema

La principal hipótesis de trabajo es que el tipo de aserciones de regresión utilizadas en testing de regresión, ya sean aserciones de test o aserciones de contratos, puede tener un impacto considerable en la efectividad del análisis para detectar defectos producidos por cambios en el software.

El objetivo principal de este trabajo es el estudio y análisis de la efectividad de distintas aserciones de regresión para detectar errores durante la tarea de testing de regresión. Esencialmente, se desea analizar la efectividad de las aserciones de test en comparación con las aserciones de contratos. Para esta tarea, se propone, para diversos casos de estudio de la literatura, simular escenarios de testing de regresión utilizando, por un lado, aserciones de test, y por otro, aserciones de contratos, generadas automáticamente.

En particular se pone el foco en las aserciones generadas por herramientas del estado del arte de generación automática de tests como Randoop[0] y Daikon[0].

A modo de ejemplo consideremos la implementación de nodo de listas doblemente enlazadas con su método insertRight en el lenguaje de programación Java.

```
 9  public class DoublyLinkedListNode {
10
11      /**
12       * Left neighbour
13       */
14      private DoublyLinkedListNode left;
15
16      /**
17      * Right neighbour
18      */
19      private DoublyLinkedListNode right; // -- Right neighbor.
20
21      /**
22       * Creates a singleton node.
23       */
24      public DoublyLinkedListNode() {
25          left = this;
26          right = this;
27      }
```

Figura 1.1: DoubleLinkedListNode

```
33      /**
34       * Inserts node `n' to the right of the current node.
35       */
36      public void insertRight(DoublyLinkedListNode n) {
37          if (n == null) throw new IllegalArgumentException("undefined for null parameter");
38          if (n.left != n) throw new IllegalArgumentException("parameter is not singleton");
39
40          DoublyLinkedListNode r = right;
41          n.setRight(r);
42          n.setLeft(this);
43          r.setLeft(n);
44          setRight(n);
45      }
```

Figura 1.2: DoubleLinkedListNode.insertRight

Supongamos ahora que es necesario modificar parte del sistema, y antes de hacerlo sería útil tener aser-

ciones que ayuden a detectar posibles errores que puedan ser insertados durante la actualización con el fin de preservar el comportamiento del sistema. Para ello se podría considerar automáticamente aserciones de test y de contratos. En la siguiente imágen se puede ver un caso de test generado por Randoop y un contrato generado por Daikon.

```
@Test
public void test0008() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "RegressionTest0.test0008");
    examples.DoublyLinkedListNode doublyLinkedListNode0 = new examples.DoublyLinkedListNode();
    examples.DoublyLinkedListNode doublyLinkedListNode1 = new examples.DoublyLinkedListNode();
    doublyLinkedListNode0.insertRight(doublyLinkedListNode1);
    doublyLinkedListNode1.remove();
    doublyLinkedListNode1.remove();
    examples.DoublyLinkedListNode doublyLinkedListNode5 = new examples.DoublyLinkedListNode();
    doublyLinkedListNode5.remove();
    examples.DoublyLinkedListNode doublyLinkedListNode7 = new examples.DoublyLinkedListNode();
    doublyLinkedListNode5.insertRight(doublyLinkedListNode7);
    examples.DoublyLinkedListNode doublyLinkedListNode9 = new examples.DoublyLinkedListNode();
    examples.DoublyLinkedListNode doublyLinkedListNode10 = new examples.DoublyLinkedListNode();
    doublyLinkedListNode9.insertRight(doublyLinkedListNode10);
    doublyLinkedListNode10.remove();
    doublyLinkedListNode7.insertRight(doublyLinkedListNode10);
    // The following exception was thrown during execution in test generation
    try {
        doublyLinkedListNode1.insertRight(doublyLinkedListNode7);
        org.junit.Assert.fail("Expected exception of type java.lang.IllegalArgumentException;\
            message: parameter is not singleton");
    } catch (java.lang.IllegalArgumentException e) {
        // Expected exception.
    }
}
```

```
DoublyLinkedListNode:::OBJECT
this == this.left.right
this.left != null
this.right != null
====================================
DoublyLinkedListNode.insertRight:ENTER
n == n.left
n == n.right
n != null
====================================
DoublyLinkedListNode.insertRight:EXIT
this.left == n.left.left
this.right == \old(n)
n.left == \old(this)
n.right == \old(this.right)
n.right == \old(this.right.left.right)
n.right == \old(this.right.right.left)
n.left != null
n.right != null
n.right.right != null
```

Figura 1.3: Test y contrato generados automáticamente por Randoop y Daikon respectivamente para el método insertRight

De esta manera, sin demasiado esfuerzo, se han generado automáticamente aserciones que podrían ayudarnos a detectar errores introducidos durante la actualización del sistema. Pero, ¿qué tan efectivos son estas aserciones para detectar errores?. O incluso, ¿cuál tipo de aserción es más eficiente?.

Para dar respuesta a estas preguntas se propone simular escenarios de testing de regresión, introduciendo defectos en el código, técnica conocida como Pruebas de mutación, para evaluar la efectividad de las aserciones generadas.

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.

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. Generación automática de aserciones

2.1.1. Técnicas estáticas

2.1.2. Técnicas dinámicas

Entre las herramientas de generación de aserciones con técnicas dinámicas podemos encontrar ...

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.

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. Generación automática de contratos

Entre las herramientas de generación automática de contratos ...

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.

SpecFuzzer es una técnica que se utiliza para...

2.3. 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
            empty.
30      * @observer // annotation added by Jeremy
31      */
32     public Object top( )
33     {
34         if( isEmpty( ) )
35             return null;
36         Object result = theArray[ topOfStack ];
37         assert(true);
38         return result;
39     }
40 }
```

Código 2.1: Ejemplo StackAr.top

2.3.1. 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

2.3.2. 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 "muertoz 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 ==¿ a - b
- COR (Conditional Operator Replacement), el cual reemplaza operadores condicionales por otros compatibles. Ej: a —— b ==¿ a b
- LOR (Logical Operator Replacement), el cual reemplaza operadores lógicos por otros compatibles. Ej: a ^{b} ==> $a|bROR(RelationalOperatorReplacement), elcualreemplazaoperadoresrelacionalesporotroscompatibles.E$ $a == b ==> a >= b$

- SOR (Shift Operator Replacement), el cual reemplaza operadores de desplazamientos de bits por otros compatibles. Ej: a ¿¿ b ==¿ a ¡¡ b
- ORU (Operator Replacement Unary), el cual reemplaza operadores unarios por otros compatibles. Ej: -a ==¿ a
- LVR (Literal Value Replacement), el cual reemplaza un literal por un valor por defecto. Ej: val = 0 ==¿ 1 and -1
- EVR (Expression Value Replacement), el cual reemplaza una expresión por un valor por defecto. Ej: return a ==¿ 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:theArra
      |==> 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.

```
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.

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

```

```
# Second step: obtain the dtrace file with Chicory
echo '-->␣Going␣to␣run␣Chicory'
java -cp \
 ${tests_bin}:${daikon_path}/daikon.jar:${subject_jar} \
 daikon.Chicory --comparability-file=\
 ${output_dir}daikon/RegressionTestDriver.decls-DynComp \
 --output-dir=${output_dir}daikon/ ${package}.RegressionTestDriver

# Third step: actual Daikon execution to infer specs
echo '-->␣Going␣to␣run␣Daikon'
java -cp ${daikon_path}/daikon.jar daikon.Daikon \
 --ppt-select-pattern \
 ${package}'\.'$2':::CLASS|'${package}'\.'$2':::OBJECT|\
␣'${package}'\.'$2'\.'$method_without_args \
 -o ${output_dir}daikon/res.inv.gz \
 ${output_dir}daikon/RegressionTestDriver.dtrace.gz

# Process result and print the Daikon specs for better readability. First
# the ones related to class invariants, and then the ones related to the
    current method
java -cp ${daikon_path}/daikon.jar daikon.PrintInvariants \
 ${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.

```
$major/bin/javac -cp $subject_jar -nowarn -J-Dmajor.export.mutants=true
    -XMutator:ALL \
 -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 erroneas. 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.

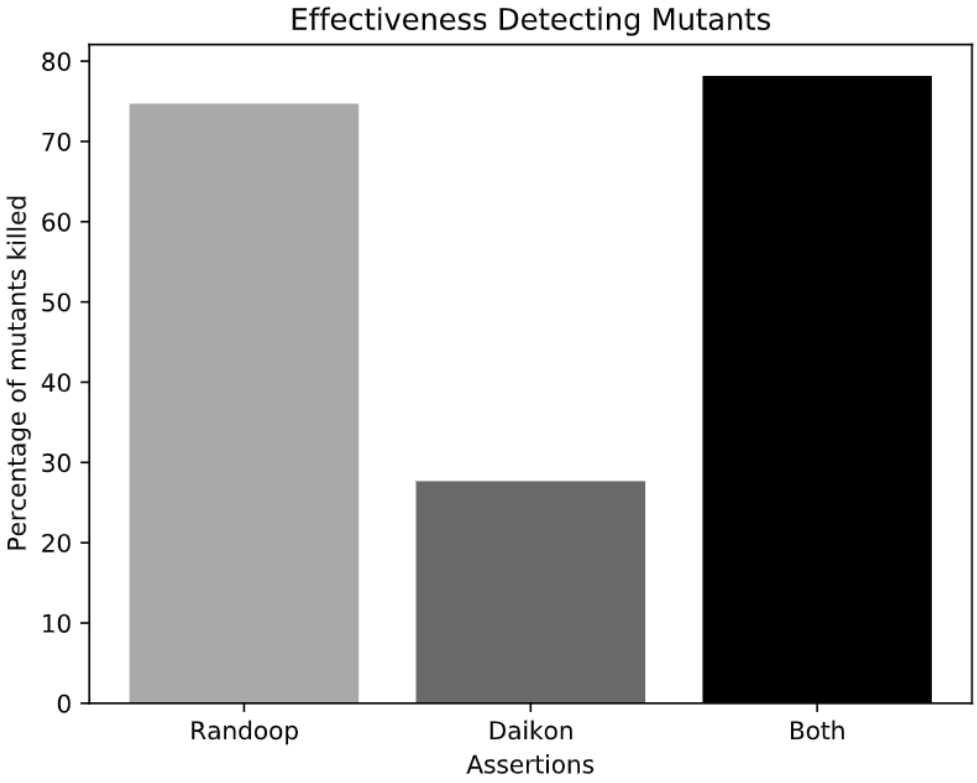

Figura 4.1: Resultados obtenidos

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

Se puede observar de los resultados obtenidos que las aserciones generadas por Randoop detectaron el 74.7 %, mientras que Daikon detectó el 27.67 %.

El total de aserciones generadas por Randoop es de 31030. Mientras que Daikon generó 5648 contratos.

Algunos casos particulares donde Daikon supera a Randoop. SimpleMethods$_{i}ncrementNumberAtIndexComposite.ad$

Randoop con 2203 aserciones detectó 4 de 18 mutantes. Mientras que Daikon detectó 16 de 18 mutantes.

Contrato generado por daikon ==========
lang3.BooleanUtils.toBoolean(java.lang.Integer, java.lang.Integer, java.lang.Integer):::ENTER value == fal-
seValue value != null trueValue != null ==========
lang3.BooleanUtils.toBoolean(java.lang.Integer, java.lang.Integer, java.lang.Integer):::EXIT == false

Mutantes: 1:LVR:51:false —==¿ true 2:ROR:52:value == null —==¿ false 3:ROR:53:trueValue == null —==¿ false 4:LVR:54:true —==¿ false 5:STD:54:result = true —==¿ ¡NO-OP¿ 6:ROR:55:falseValue == null —==¿ false 7:LVR:56:false —==¿ true 8:STD:56:result = false —==¿ ¡NO-OP¿ 9:COR:58:value.equals(trueValue) —==¿ false 10:COR:58:value.equals(trueValue) —==¿ true 11:LVR:59:true —==¿ false 12:STD:59:result = true —==¿ ¡NO-OP¿ 13:COR:60:value.equals(falseValue) —==¿ false 14:COR:60:value.equals(falseValue) —==¿ true 15:LVR:61:false —==¿ true 16:STD:61:result = false —==¿ ¡NO-OP¿ 17:LVR:66:true —==¿ false 18:EVR:67:result —==¿ false

Randoop detectó mutantes 2, 6 y 7. Daikon no detectó 5 y 9

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.