Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Generación Automática de Casos de Test para EPAs: Un enfoque basado en Algoritmos Genéticos

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Pablo Antonio

Director: Juan Pablo Galeotti

Codirector:

Buenos Aires, 2017

GENERACIÓN AUTOMÁTICA DE CASOS DE TEST PARA EPAS: UN ENFOQUE BASADO EN ALGORITMOS GENÉTICOS

Hay ciertos tipos de programas que poseen requerimientos no triviales con respecto al orden en que se ejecutan las distintas partes de su código para producir determinados resultados; APIs, interfaces gráficas o servidores web son ejemplos de estos. Además, existen formas de representar los distintos estados de estos componentes y las acciones que deben ocurrir para que estos pasen de un estado a otro, como las EPAs (Enabledness Preserving Abstractions).

Existe investigación previa que sugiere que la alta cobertura de la EPA de un componente, por parte de una test suite, es un buen predictor del número de fallas que el test suite puede llegar a encontrar[1]. El mismo trabajo sugiere que la obtención de mayores coberturas de EPA no implica, necesariamente, test suites más grandes. Estos resultados invitan a pensar que una herramienta de generación automática de casos de test que tuviera en cuenta la EPA del componente a probar podría ser muy efectiva y útil.

En esta tesis nos proponemos adaptar la herramienta EvoSuite para que esta tenga en cuenta la EPA asociada al componente a la hora de generar casos de test para el mismo. A partir de esta herramienta adaptada, nos proponemos realizar experimentos con diferentes sujetos (clases) con el fin de evaluar la eficiencia y la eficacia de esta en generar conjuntos de test con alta cobertura de EPA, y su impacto en otras métricas como la cobertura de código (líneas y ramas). Además, buscaremos corroborar la hipótesis de que altas coberturas de EPA se condicen con una mayor efectividad en la detección de fallas.

Como último objetivo, nos proponemos explorar el uso de esta herramienta en la detección de incongruencias entre el código y su modelo de EPA asociado.

Palabras claves: testing de software, generación de casos de test, protocolos, API, EvoSuite.

Índice general

1. INTRODUCCIÓN

El testing de software puede definirse como "cualquier actividad que tenga como objetivo evaluar un atributo o capacidad de un programa o sistema y a determinar que cumple con los resultados requeridos"[2]. En ese sentido, puede decirse que es una práctica que encuentra sus orígenes junto a la aparición misma de las primeras computadoras, o incluso antes. Hay referencias al testing de software, en la literatura, desde al menos 1950[2].

Pese a los cambios que han habido en las computadoras y en la práctica de la programación en las últimas décadas, el testing de software sigue siendo una parte importante de cualquier proyecto de software[3] y es, hoy en día, una de las prácticas más usadas para comprobar y, en última instancia, mejorar la calidad del software[4].

1.1. Búsqueda de errores

Una forma de encarar el testing de software es a partir de la búsqueda de errores en los programas[3]. Si lo que se busca es poner en evidencia un error, un modo de hacerlo es ejercitando el programa para una entrada determinada y observando que, frente a esta entrada, el mismo no se comporta como era esperado. Lo anterior suele denominarse en la bibliografía un caso de test[5], es decir, una entidad compuesta por:

- Una entrada para el programa a probar
- Un *oráculo*, es decir, un procedimiento que verifica que el programa se comporta como es esperado frente a la entrada provista

En este caso, cuando hablamos de entrada nos estamos refiriendo a un conjunto de datos y operaciones que, en conjunto, ejercitan el programa que se busca probar.

Si nuestro objetivo es descubrir todos los errores de un programa, entonces podríamos pensar en tener casos de test para todas las entradas posibles de este y un oráculo capaz de decidir si cada uno de los comportamientos observados fue el deseado. De esa forma, podríamos identificar absolutamente todos los errores del programa y arreglarlos. Aquí es donde se hace presente una observación empírica: Lo anterior suele no ser práctico o, incluso, posible en la realidad[3], aún cuando se trata de programas triviales.

A la observación anterior le sigue pensar que sólo queda, como posibilidad, tratar de encontrar la mayor cantidad de errores en los programas en el tiempo del que se dispone. Este tiempo, por supuesto, dependerá de cada proyecto, pero en la bibliografía hay estimaciones que indican que, unos años atrás, el testing de software representaba la mitad del tiempo dedicado a producir un programa, o incluso más[6].

Sucede que hacer buen testing de software, de forma manual, es difícil[3], lleva tiempo y dinero, y puede tornarse en una actividad tediosa y propensa a errores[5]. En este contexto, con el objetivo de mejorar la eficacia y la eficiencia de los mecanismos de testing de software, han surgido distintas formas de automatizar los procesos involucrados[7]. Uno de estos procesos, quizás uno de los más importantes y al que nos abocamos en esta tesis, es el de la generación de casos de test.

1.2. EvoSuite

EvoSuite es una herramienta que es capaz de generar casos de test para clases escritas en el lenguaje Java. Para esto, usa un acercamiento novedoso en el que optimiza conjuntos enteros de casos de test (test suites) buscando satisfacer uno o varios criterios de cobertura. Además, facilita la tarea manual posterior de oráculo, reduciendo al mínimo la cantidad de casos de test y sugiriendo pequeñas aserciones que permiten al desarrollador detectar desviaciones respecto del comportamiento esperado del programa.

EvoSuite es considerada una herramienta de punta en lo que refiere a generación automática de tests unitarios; ha ganado, en varias oportunidades, la competencia de herramientas de testing del Workshop sobre "Search-based Software Testing"[8]. Sin embargo, tanto EvoSuite como otras herramientas, aún tienen que mejorar para poder ser consideradas herramientas confiables para la búsqueda de errores[9], por lo que se trata de un área con mucho trabajo aún por delante.

1.3. *Enabledness-Preserving Abstractions (EPAs)*

Hay ciertos tipos de programas que poseen requerimientos no triviales con respecto al orden en que se ejecutan las distintas partes de su código para producir determinados resultados; APIs, interfaces gráficas o servidores web son ejemplos de estos. Además, existen formas de representar los distintos estados de estos componentes y las acciones que deben ocurrir para que estos pasen de un estado a otro, como las EPAs (Enabledness Preserving Abstractions)[10].

1.4. Esta tesis

Existe investigación previa que sugiere que la alta cobertura de la EPA de un componente, por parte de una test suite, es un buen predictor del número de fallas que el test suite puede llegar a encontrar, así como de la cobertura de código (líneas y ramas) que puede alcanzar[1]. Además, El mismo trabajo sugiere que la obtención de mayores coberturas de EPA no implica, necesariamente, test suites más grandes. Estos resultados invitan a pensar que una herramienta de generación automática de casos de test que tuviera en cuenta la EPA del componente a probar podría ser muy efectiva y útil. Sin embargo, no existe, hasta donde conocemos, una herramienta de generación automática de tests que busque explícitamente sacar provecho de los resultados mencionados.

En esta tesis nos proponemos adaptar la herramienta EvoSuite para que esta tenga en cuenta la EPA asociada al componente a la hora de generar casos de test para el mismo. A partir de esta herramienta adaptada, nos proponemos realizar experimentos con diferentes sujetos (clases) con el fin de evaluar la eficiencia y la eficacia de esta en generar conjuntos de test con alta cobertura de EPA, y su impacto en otras métricas como la cobertura de código (líneas y ramas). Además, buscaremos corroborar la hipótesis de que altas coberturas de EPA se condicen con una mayor efectividad en la detección de fallas.

Como último objetivo, nos proponemos explorar el uso de esta herramienta en la detección de incongruencias entre el código y su modelo de EPA asociado.

2. PRELIMINARES

2.1. Generación de casos de test mediante métodos de búsqueda

El problema de la generación de casos de test puede ser expresado como un problema de búsqueda u optimización: Dados todos los posibles conjuntos de casos de test que pueden ser generados para un sistema (o sujeto de prueba) en particular, el objetivo es encontrar el mejor[1] conjunto o, al menos, uno suficientemente bueno.

Para que la anterior definición del problema tenga sentido, es necesaria alguna forma de comparar o medir conjuntos de casos de test. En ese sentido, existen métricas que, combinadas o aisladas, pueden servir de criterio a la hora de comparar conjuntos de casos de test. Dos de las métricas más utilizadas son las conocidas como **métricas de cobertura de código**: la métrica de cobertura de líneas y la métrica de cobertura de ramas.

Definición 2.1.1. Definimos la métrica de **cobertura de líneas** de un conjunto de casos de test, con respecto a un sujeto a probar, como el porcentaje, del total de las líneas de código del sujeto, que fueron ejercitadas al ejecutar el conjunto de casos de test.

Definición 2.1.2. Definimos la métrica de **cobertura de ramas** de un conjunto de casos de test, con respecto a un sujeto a probar, como el porcentaje, del total de las ramas en el código del sujeto, que fueron ejercitadas al ejecutar el conjunto de casos de test. Se entiende por rama a cada una de las bifurcaciones posibles en el código a causa de estructuras de control presentes en él.

En la práctica, independientemente de las métricas que se elijan, el espacio de búsqueda del problema antes descripto suele ser muy grande. Además, debido en parte a la presencia de estructuras de control de flujo, como las condicionales o los bucles, estrategias de búsqueda sencillas pueden no ser suficientes para obtener buenos resultados en tiempos aceptables. En esos casos, pueden ser necesarias otro tipo de estrategias como las metaheurísticas de búsqueda[11].

En general, cuando se habla de generación de casos de test mediante métodos de búsqueda, se está haciendo referencia al uso de estrategias que se apoyan en metaheurísticas de búsqueda[12].

2.2. *Whole test suite generation*

No es inusual que los requisitos que se le exijan al conjunto de casos de test, para poder considerarlo aceptable, sean varios. Por ejemplo, puede pedirse que el conjunto de casos de test que buscamos tenga una buena cobertura de líneas y también una buena cobertura de ramas. En el contexto de los métodos de búsqueda, se puede pensar en estos requisitos en términos de objetivos a cumplir. Por ejemplo, para el requisito de obtener una buena cobertura de líneas, el algoritmo podría definir cada línea en el código del sujeto a probar como un objetivo que busca cumplir, en este caso, ejecutando la línea.

Una estrategia común en la literatura es generar un caso de test por cada objetivo a cumplir. Este acercamiento presenta algunos problemas. Por un lado, es común que

[1] O, de haber varios, uno de los mejores.

algunos objetivos sean más difíciles de cumplir que otros. Incluso, pueden existir objetivos que sean imposibles de cumplir. El tiempo destinado a cumplir esos objetivos imposibles es, por definición, tiempo perdido, y detectar si un objetivo es factible puede ser un problema indecidible[13]. La existencia de objetivos que son más difíciles de cumplir que otros, independientemente de si esta dificultad puede ser fácilmente estimada o no[2], lleva al problema de cómo administrar el tiempo total del que se dispone para la búsqueda a cada uno de los objetivos a cumplir y cómo redistribuir ese tiempo cuando un objetivo es cumplido antes del tiempo que se le otorgó.

Otra dificultad que surge de esta estrategia es que es difícil predecir el tamaño del conjunto de casos de test resultante, debido al fenómeno de la "cobertura colateral", esto es, la existencia de objetivos que, al cumplirse, hacen que otros objetivos se cumplan implícitamente. Por ejemplo, puede darse que, para poder ejecutar una línea del código necesariamente haya que haber pasado por otra antes. A pesar de que han habido intentos de explotar la cobertura colateral para optimizar la generación de tests, no conocemos una evaluación concluyente de su efectividad. El tamaño del conjunto de casos de test es importante porque de ello dependerá, entre otras cosas, el esfuerzo necesario para realizar el posterior trabajo manual de oráculo.

La técnica de "whole test suite generation"[13] (generación de conjuntos de casos de test de forma completa) propone utilizar una técnica evolutiva en la que, en lugar de evolucionar cada caso de test por separado, se evolucionan todos los casos de test de un conjunto de casos de test al mismo tiempo, y todos los objetivos a cumplir son considerados simultáneamente. Existe evidencia de que esta técnica permite conseguir una mayor cobertura general que la técnica tradicional que se enfoca en los objetivos de forma individual, y resultados que apoyan la validez y efectividad de esta técnica en la generación automática de casos de test[15].

2.3. EvoSuite

EvoSuite es una herramienta de generación de casos de test para el lenguaje Java que aplica la técnica de "whole test suite generation". El funcionamiento de EvoSuite está basado en un *algoritmo genético*.

2.3.1. Algoritmo genético de EvoSuite

Los algoritmos genéticos son una metaheurística muy utilizada en el ámbito de la optimización que está inspirada en el proceso de selección natural. La idea de esta clase de algoritmos es conseguir una solución exacta o aproximada al problema de optimización al que se aplican, partiendo de una población inicial y aplicando repetidos pasos de evolución que involucran operaciones inspiradas en la naturaleza. En la jerga de los algoritmos genéticos, cada individuo de la población es también conocido como "cromosoma". Algunas de las operaciones que se aplican para acercar al algoritmo a una solución son:

- **Mutación:** La alteración de los cromosomas con el objetivo de mantener una "diversidad genética".

[2] Existe trabajo, en la literatura, orientado a predecir la dificultad de cumplimiento de objetivos en código procedural[14], pero su evaluación y utilidad en software orientado a objetos es aún una pregunta de investigación abierta.

- **Recombinación (o *crossover*):** Inspirada en la reproducción, permite producir una solución "hija" a partir de más de un "padre".
- **Selección:** Basada en la idea de aptitud, los mejores individuos son seleccionados para dar lugar a la siguiente generación.

El algoritmo genético de EvoSuite parte de una población inicial aleatoria de cromosomas (conjuntos de casos de test) y realiza pasos de evolución de esta población hasta (i) encontrar una solución, es decir, un conjunto de casos de test que satisfaga un criterio especificado (por ejemplo, cubrimiento de todas las líneas del código del sujeto), o bien (ii) hasta agotar los recursos de los que se dispone (por ejemplo, tiempo total o número de evoluciones).

Siguiendo la idea de los algoritmos genéticos, en cada paso de evolución, EvoSuite crea una nueva generación con los mejores individuos de la generación anterior (proceso conocido como "elitismo"). Luego, hasta no conseguir una población del tamaño de la generación anterior, se llevan adelante los siguientes pasos:

1. Se toman dos soluciones "padres", P_1 y P_2 mediante selección por ranking, esto es, un mecanismo que da más chances de ser selccionados a los individuos más aptos, pero de una forma justa con respecto al resto de los individuos.
2. Se realiza una recombinación entre estos padres con una probabilidad determinada, dando por resultado dos soluciones hijas O_1 y O_2,
3. Se realiza la mutación de O_1 y O_2.
4. Dependiendo de condiciones sobre aptitud y tamaño, los padres o los hijos pasan a formar parte de la nueva generación.

2.3.2. Criterios de cobertura

La clave para que este algoritmo conduzca a soluciones interesantes recae en las funciones de aptitud (o *fitness*) que se utilicen. En EvoSuite, las funciones de aptitud están determinadas por el **criterio de cobertura** seleccionado. Algunos ejemplos de estos son:

1. `LINE`: Criterio asociado a una función de aptitud que prefiere individuos que se acerquen a ejecutar una mayor cantidad de líneas en el sujeto a probar. Este criterio está pensado para obtener buena cobertura de líneas.
2. `BRANCH`: Criterio asociado a una función de aptitud que prefiere individuos que estén más cerca de ejecutar una mayor cantidad de ramas. Este criterio está pensado para obtener una buena cobertura de ramas sobre el código del sujeto.

La idea de cercanía en ambos criterios se desprende del hecho de que sus funciones de aptitud asociadas tienen en cuenta una noción de "distancia" a cubrir todos los objetivos. Por ejemplo, en el caso de la función de aptitud del criterio `BRANCH`, se involucra el concepto de "distancia a una rama", una estimación de qué tan lejos está un predicado de satisfacer la condición booleana (falsa o verdadera) que permitirá ejecutar una rama. Así, la función de aptitud es una estimación de qué tan lejos está el conjunto de casos de test en cuestión de satisfacer las condiciones de todas las ramas. La función de aptitud del criterio `LINE` aplica una idea parecida[16].

Además de la posibilidad de elegir un único criterio de cobertura para orientar la generación de casos de test, EvoSuite permite la utilización de criterios combinados. Por ejemplo, puede ejecutarse EvoSuite utilizando el conjunto de criterios {`LINE`, `BRANCH`}. Los criterios utilizados en estas combinaciones no deben ser conflictivos entre sí.

Definición 2.3.1. Dos criterios de cobertura son **conflictivos entre sí** si puede darse el caso de que, al agregar un caso de test a un conjunto para incrementar la cobertura de un criterio, se puede afectar negativamente la cobertura del otro.

Un ejemplo trivial de esto podría ser un conjunto que incluye, además del criterio `LINE`, un criterio `NOLINE` que busca maximizar la cantidad de líneas no ejecutadas. Si bien este es un caso evidente, podrían darse otros en los cuales la conflictividad no sea inmediata, por lo que debe tenerse en cuenta esta cuestión a la hora de elegir combinaciones de criterios.

Cuando se combinan varios criterios, la función de aptitud asociada se calcula en base a la suma de los resultados de las funciones de aptitud de cada uno de los criterios involucrados.

Definición 2.3.2. Sean $f_1, \dots f_n$ funciones de aptitud de los n criterios involucrados, la **funcion de aptitud para el criterio combinado**, f_{comb}, se define:

$$f_{comb}(c) = \sum_{i=1}^{n} (f_i(c))$$

para todo conjunto c de casos de test (individuo).

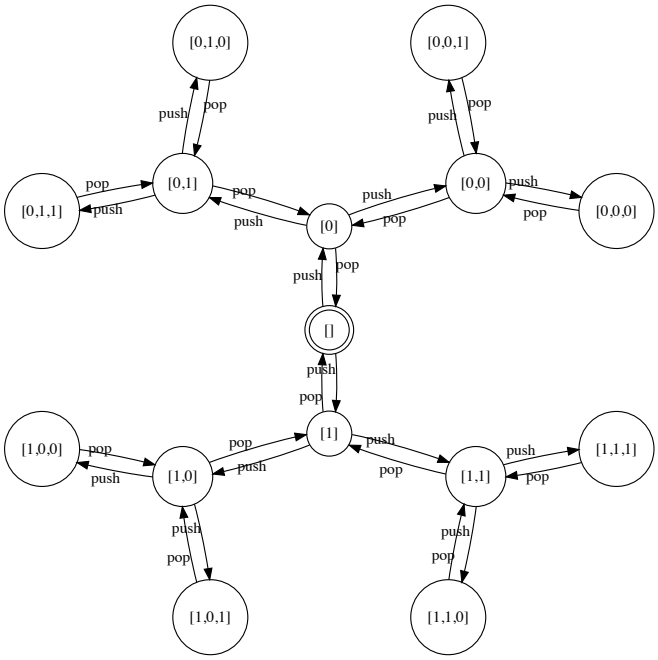
Existe investigación previa que explora el impacto de la combinación de criterios tanto en el tamaño del conjunto de casos de test generado como en la cobertura alcanzada por cada uno de los criterios que constituyen la combinación[16]. Los resultados de esta investigación muestran que, por un lado, al usar un número grande (nueve) de criterios en la combinación, el tamaño del conjunto de casos de test crece sustancialmente, aunque no a un punto inmanejable por desarrolladores, y, por otro, que la cobertura individual de los criterios involucrados en las combinaciones decrece muy poco (0.4 % en promedio), y en algunos casos hasta se da que algunas combinaciones potencian las coberturas individuales de los criterios.

2.4. *Enabledness-preserving abstractions (EPAs)*

Una forma de expresar el protocolo de una clase (del paradigma orientado a objetos) es mediante una máquina de estados. Existe un estado inicial para los objetos de la clase y, a partir de este, se efectúan operaciones que llevan a los objetos a otros estados. No todas las operaciones pueden efectuarse en todos los estados; por ejemplo, no puede desapilarse un elemento de una pila que se encuentra vacía.

Siguiendo con el ejemplo de la pila, pensemos en una que sólo puede albergar los valores enteros 0 y 1, y con una capacidad máxima fijada en 3. Se puede pensar que la pila, al momento de su construcción, se encuentra en un estado inicial que representa la pila vacía: []. Al agregar el elemento 0, la pila pasa a un segundo estado, el estado que representa a la pila conteniendo ese único elemento: [0]. Si luego se agrega un 1, la pila pasará a un estado $[0, 1]$. A su vez, si se desapila el último elemento agregado, la pila volverá a estar en el estado [0]. Cuando la pila se encuentra en alguno de los estados que representan a

Fig. 2.1: Máquina de estados que representa el protocolo de una pila de capacidad máxima 3 y valores permitidos 0 y 1.



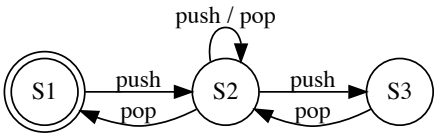
la pila llena (con 3 elementos), sólo la operación de desapilar se encuentra disponible. La figura 2.1 muestra los estados y posibles transiciones en una pila como la descripta.

Como se ve, el número de estados posibles dependerá de la capacidad de la pila y de la cantidad de elementos distintos que puedan ser agregados a la pila. En este ejemplo, la capacidad está restringida a un número muy bajo, y sólo se permiten, como elementos, los números 0 y 1. En un caso más real, con una capacidad fijada en un valor grande, y permitiendo cualquier entero representable como posible elemento, la máquina de estados se volvería muy grande. Incluso, para algunas estructuras, el número de estados podría no ser finito (por ejemplo, para una pila no acotada).

Las *enabledness-preserving abstractions* (EPAs) [10] son abstracciones que permiten representar el espacio potencialmente infinito de estados y transiciones entre estos de un objeto de una clase (en el sentido de la programación orientada a objetos) en un conjunto finito de clases (en el sentido de clases de equivalencia) de estados y sus transiciones. Podría decirse que las EPAs pueden representar de manera compacta el protocolo de una clase. La idea fundamental detrás del modelo de EPAs es que los estados pueden agruparse según las operaciones que pueden realizarse a partir de ellos. Por ejemplo, para el caso de la pila antes mencionada, los estados [0] y [1] son equivalentes, en tanto que ambos permiten las mismas operaciones: Apilar y desapilar. No son equivalentes, sin embargo, al estado [], que sólo permite la operación de apilar elementos. La figura 2.2 muestra la EPA correspondiente a la estructura que hemos usado como ejemplo.

En sentido estricto, las EPAs pueden describirse como un *labeled transition system (LTS)* (o "sistema de transiciones etiquetado") finito y potencialmente no determinístico.

Fig. 2.2: EPA de una pila con capacidad máxima (*bounded stack*).



Definición 2.4.1. Definimos una **enabledness-preserving abstraction (EPA)** M como $M = \langle \Sigma, S, S_0, \delta \rangle$, donde $\Sigma = \{m_1, \ldots m_n\}$ son nombres de métodos (acciones), S es un conjunto de estados, $S_0 \in S$ es el estado inicial y $\delta \subseteq S \times \{m_1, \ldots, m_n\} \times S$ es la relación de transición.

Un método m_j se dice **enabled (permitido)**, para un estado S_i si y sólo si $(S_i, m_j, S_k) \in \delta$ para algún estado S_k.

Es importante notar que cualquier ejecución de código válida, es decir, que utilice a un objeto de la clase correctamente en términos de su protocolo, representará un camino en la EPA de la clase. Por ejemplo, si tuviéramos un código que parte de la pila vacía, le agrega dos elementos, y luego desapila el último, el camino sería:

$$S_1 \xrightarrow{push} S_2 \xrightarrow{push} S_2 \xrightarrow{pop} S_2$$

3. GENERANDO CASOS DE TEST USANDO EPAS

3.1. Sujeto experimental de ejemplo

Para entender mejor las siguientes secciones, un caso particular puede ser de ayuda. Usaremos el caso de la pila de capacidad máxima fija, con las mismas operaciones y la misma semántica que en el capítulo anterior, aunque con la capacidad fijada en un valor arbitrario (en este caso, 10). El código 3.1 representa un extracto de lo que podría ser el código de esta estructura.

Código 3.1: Código de una pila de capacidad máxima fija

```
public class BoundedStack {
        private final static int DEFAULT_SIZE = 10;
        private final Object[] elements = new Object[DEFAULT_SIZE];
        private int index = -1;

        public BoundedStack() { ... }
        public void push(Object object) { ... }
        public Object pop() { ... }
}
```

La figura 3.1 es la EPA correspondiente a esta estructura. Notar que se trata de una representación similar a la dada en la figura 2.1, con el agregado de un estado inicial que representa al objeto antes de su construcción.

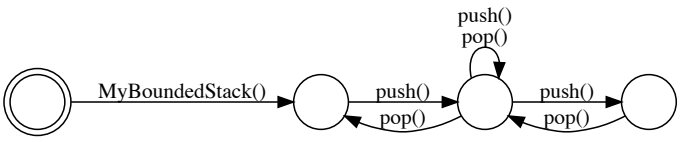
3.2. Representación de EPAs

Para poder llevar a cabo la generación de casos de test para un sujeto de prueba sacando provecho de su EPA, hace falta contar con una representación de la misma. Nuestra herramienta toma como entrada la EPA del sujeto especificada mediante un formato sencillo basado en XML – el mismo que utiliza, por omisión, la herramienta Contractor [17].

Esencialmente, el formato permite describir:

- Cada uno de los estados de la EPA, especificados mediante la etiqueta `<state>`
- Los métodos permitidos en cada uno de los estados de la EPA

Fig. 3.1: EPA correspondiente a *BoundedStack*



- Las transiciones de la EPA, mediante la etiqueta `<transition>`
- El estado inicial de la EPA

Esta información permite expresar a una EPA unívocamente, según la definición 2.4.1.

Para el caso de la pila de capacidad máxima fija, la representación XML de la EPA podría ser la explicitada en el código 3.2.

3.3. Relación entre EPA y código del sujeto

Para poder relacionar el código a la EPA, se introdujo dos anotaciones (*annotations*) del lenguaje Java:

- `org.evosuite.epa.EpaState`: Esta anotación se utiliza en métodos booleanos especiales que permiten a EvoSuite decidir en qué estado de la EPA se encuentra el objeto en cuestión. Su uso es sencillo: simplemente se anota el método especificando su estado asociado. Por ejemplo, el método que decide si el objeto se encuentra en el estado `S1` es anotado con `@EpaState(name = "S1")`.
- `org.evosuite.epa.EpaAction`: Esta anotación permite asociar al método señalado con una acción en la EPA. Por ejemplo, el método que realiza la acción de `push` en la EPA puede ser señalado con la anotación `@EpaAction(name = "push")`.

El código 3.3 representa el código original de la pila instrumentado para poder ser utilizado por nuestro prototipo.

Para el caso de esta estructura, podríamos decir que el estado en que el objeto se encuentra dependerá, básicamente, de si este está vacío (`S1`), lleno (`S3`) o ninguno de los dos (`S2`). Cuando la pila está vacía, su operación de `pop()` está desactivada, y cuando está llena su operación de `push()` lo está. Si la pila no se encuentra ni llena ni vacía, las dos operaciones están permitidas. El código 3.4 muestra una posible implementación de los métodos booleanos anotados con la anotación `org.evosuite.epa.EpaState`.

3.4. Extendiendo EvoSuite

Como se explica en la sección 2.3, el funcionamiento de EvoSuite está basado en un algoritmo genético que, como parte de la evaluación de la aptitud de los individuos (conjuntos de casos de test), hace uso de funciones de aptitud. Estas funciones están determinadas por el criterio de cobertura seleccionado.

Como parte de esta tesis, se definieron los criterios de cobertura descriptos a continuación.

Definición 3.4.1. Definimos el criterio `EPATRANSITION` como el criterio asociado a la función de aptitud que prefiere conjuntos de casos de test que ejerciten una cantidad mayor de transiciones de EPA en el sujeto.

Definición 3.4.2. Definimos el criterio `EPAERROR` como el criterio asociado a la función de aptitud que prefiere conjuntos de casos de test que ejerciten una cantidad mayor de transiciones inválidas de EPA (es decir, transiciones que no están contempladas en la EPA) en el sujeto.

Código 3.2: Representación XML de la EPA de una pila de capacidad máxima fija

```
<?xml version="1.0" encoding="utf-8"?>
<abstraction initial_state="S0" input_format="code-with-pre"
             name="BoundedStack">
    <label name="BoundedStack()"/>
    <label name="push()"/>
    <label name="pop()"/>

    <state name="S0">
        <enabled_label name="BoundedStack()"/>
        <transition destination="S1" label="BoundedStack()"
                    uncertain="false"
                    violates_invariant="false"/>
    </state>

    <state name="S1">
        <enabled_label name="push()"/>
        <transition destination="S2" label="push()" uncertain="false"
                    violates_invariant="false"/>
    </state>

    <state name="S2">
        <enabled_label name="push()"/>
        <enabled_label name="pop()"/>
        <transition destination="S2" label="push()" uncertain="false"
                    violates_invariant="false"/>
        <transition destination="S2" label="pop()" uncertain="false"
                    violates_invariant="false"/>
        <transition destination="S3" label="push()" uncertain="false"
                    violates_invariant="false"/>
        <transition destination="S1" label="pop()" uncertain="false"
                    violates_invariant="false"/>
    </state>

    <state name="S3">
        <enabled_label name="pop()"/>
        <transition destination="S2" label="pop()" uncertain="false"
                    violates_invariant="false"/>
    </state>
</abstraction>
```

Código 3.3: Código de *BoundedStack* instrumentado

```
import org.evosuite.epa.EpaAction;
import org.evosuite.epa.EpaState;

public class BoundedStack {

        ...

        private final static int DEFAULT_SIZE = 10;
        private final Object[] elements = new Object[DEFAULT_SIZE];
        private int index = -1;

        @EpaAction(name = "MyBoundedStack()")
        public MyBoundedStack() { ... }

        @EpaAction(name = "push()")
        public void push(Object object) { ... }

        @EpaAction(name = "pop()")
        public Object pop() { ... }

        @EpaState(name = "S1")
        private boolean queryForS1() { ... }

        @EpaState(name = "S2")
        private boolean queryForS2() { ... }

        @EpaState(name = "S3")
        private boolean queryForS3() { ... }
}
```

Código 3.4: Implementación de los métodos booleanos anotados con `EpaState`

```
public class BoundedStack {

        ...

        private boolean isPushEnabled() {
                return index != elements.length - 1;
        }

        private boolean isPopEnabled() {
                return index != -1;
        }

        @EpaState(name = "S1")
        private boolean queryForS1() {
                return isPushEnabled() && !isPopEnabled();
        }

        @EpaState(name = "S2")
        private boolean queryForS2() {
                return isPushEnabled() && isPopEnabled();
        }

        @EpaState(name = "S3")
        private boolean queryForS3() {
                return !isPushEnabled() && isPopEnabled();
        }
}
```

Como se describe en [18], para agregar un nuevo criterio de cobertura a EvoSuite, debe definirse:

- Una función de aptitud para un caso de test
- Un generador de funciones de aptitud asociadas a un objetivo
- Una función de aptitud para un conjunto de casos de test

3.4.1. Función de aptitud para un caso de test

Si bien la forma de uso típica de EvoSuite se apoya en la evolución de conjuntos de casos de test (*test suites*), es necesaria la definición de una función de aptitud aplicable a un único caso de test. La razón es que EvoSuite permite la evolución de poblaciones de casos de test individuales (por ejemplo, mediante el parámetro `-generateTests`), y además precisa estas funciones para el post-procesamiento de los casos de test.

La idea es que la función de aptitud es instanciada con un objetivo de cobertura particular (por ejemplo, una transición de la EPA, o una rama del código) y, mediante su método `getFitness()`, que recibe un individuo (caso de test) y el resultado de la ejecución de un test (que incluye una cantidad de datos recabados durante la ejecución del test, incluyendo la traza de métodos ejecutados), puede calcular la aptitud del caso de test en cuestión.

En nuestro caso, definimos nuestra función de aptitud para casos de test en la clase `EPATransitionCoverageTestFitness`. En la instanciación de esta clase se fija como objetivo una transición particular en la EPA, y la implementación del método `getFitness()` simplemente resuelve si el caso de test cubre la transición asociada a la función de aptitud (valor de aptitud 0), o no (valor de aptitud 1).

3.4.2. Generador de funciones de aptitud

Como vimos, la función de aptitud a ser aplicada a un caso de test, está asociada, desde su creación, a un objetivo de cobertura. Por ejemplo, la función de aptitud `EPATransitionCoverageTestFitness` está asociada a una transición de la EPA a cubrir. Para cada criterio de cobertura se precisa, entonces, alguna forma de obtener todos los objetivos (asociados a funciones de cobertura) que se busca cubrir. Aquí es donde entra en juego el generador de funciones de aptitud.

Los generadores de funciones de aptitud heredan de la clase `AbstractFitnessFactory`. Para los criterios que introdujimos en EvoSuite en esta tesis, se crearon los generadores `EPATransitionCoverageFactory` y `EPAErrorFactory`. Estos generadores implementan el método `getCoverageGoals()` que devuelve una lista de funciones de aptitud para casos de test. El primero de estos generadores devuelve una lista de `EPATransitionCoverageTestFitness` asociados a todas las transiciones de EPA posibles para el sujeto, mientras que el último devuelve una lista análoga pero asociada a todas las transiciones que no están representadas en la EPA del sujeto.

3.4.3. Función de aptitud para un conjunto de casos de test

Como último paso para extender EvoSuite con un nuevo criterio, es necesario crear una función de aptitud que pueda aplicarse a conjuntos de casos de test en lugar de a casos

de test aislados. Para esto, debe crearse una clase que herede de `TestSuiteFitnessFunction`. En nuestra extensión de EvoSuite, esta clase es `EPASuiteFitness`.

El método más importante a redefinir aquí es `getFitness()`, que recibe un individuo (conjunto de casos de test) y calcula un valor de aptitud. Si bien los criterios `EPATRANSITION` y `EPAERROR` están asociados a objetivos distintos, la idea para el cálculo de la aptitud es la misma: Buscamos cubrir un conjunto de transiciones de EPA (válidas o no), y nuestro valor de aptitud será la diferencia entre el total de transiciones que se busca cubrir y las que efectivamente fueron cubiertas por el conjunto de casos de test en cuestión. El valor óptimo de aptitud es, entonces, cero.

Dado que los objetivos son distintos para los dos criterios que agregamos, dejamos el cálculo de la aptitud en `EPASuiteFitness`, pero encapsulamos la obtención de los objetivos en las clases `EPATransitionCoverageSuiteFitness` y `EPAErrorSuiteFitness`.

4. EVALUACIÓN

4.1. Preguntas de investigación

Existe investigación previa que sugiere que la alta cobertura de la EPA de un componente, por parte de una test suite, es un buen predictor del número de fallas que el test suite puede llegar a encontrar, así como de la cobertura de código (líneas y ramas) que puede alcanzar[1]. El mismo trabajo sugiere que la obtención de mayores coberturas de EPA no implica, necesariamente, test suites más grandes. Estos resultados invitan a pensar que una herramienta de generación automática de casos de test que tuviera en cuenta la EPA del componente a probar podría ser muy efectiva y útil. Sin embargo, no existe, hasta donde conocemos, una herramienta de generación automática de tests que busque explícitamente sacar provecho de los resultados mencionados.

Como parte del trabajo para esta tesis, se decidió adaptar la herramienta EvoSuite para que esta tuviera en cuenta la EPA asociada al componente para el que se busca generar casos de test. En particular, se decidió agregar dos nuevos criterios a los que ya dispone esta herramienta: `EPATRANSITION` y `EPAERROR`.

El objetivo de agregar estos criterios fue poder utilizarlos como guía en el mecanismo de generación de casos de test con el que cuenta la herramienta.

4.1.1. Criterios simples

Pregunta 1. *¿Cuán efectivo es el prototipo adaptado, utilizado únicamente con el criterio agregado `EPATRANSITION`, en generar test suites con buena cobertura de transiciones de EPA?*

Esta primera pregunta nos permite comprobar que nuestra herramienta, utilizando solamente el criterio agregado `EPATRANSITION` cumple el objetivo buscado de conseguir coberturas de transiciones de EPA altas.

Pregunta 2. *¿Cuán efectivo es el prototipo, utilizado únicamente con el criterio agregado `EPATRANSITION` para generar casos de test con buena detección de fallas?*

Como dijimos, la investigación previa menciona una correlación entre la alta cobertura de transiciones de EPA y la detección de fallas. Resulta de interés observar si estos resultados se reflejan en nuestra experimentación.

Pregunta 3. *¿Cuán efectivo es el prototipo, utilizado únicamente con el criterio agregado `EPATRANSITION` para generar casos de test con buena cobertura de código (líneas y ramas)?*

También se sugiere, en la investigación previa, que existe una correlación entre cobertura de las transiciones de EPA y la cobertura de código (líneas y ramas), y queremos observar si esto se manifiesta en nuestra experimentación.

Pregunta 4. *¿Cuán efectivo es el prototipo, utilizado únicamente con el criterio agregado `EPAERROR` en detectar transiciones de EPA inválidas?*

Si al ejecutar un conjunto de casos de test se detecta una transición inválida, esto podría significar que existe un error en la EPA, en el sentido de que el comportamiento del componente no está correctamente especificado en ella, pero también podría significar que el componente no cumple con una especificación correcta. En cualquiera de los dos casos, una herramienta capaz de encontrar este tipo de diferencias podría ser de mucha utilidad. [1]

4.1.2. Criterios combinados

Como mencionamos anteriormente, existe investigación previa que muestra que coberturas individuales pueden ser potenciadas mediante la combinación de criterios no conflictivos entre sí [16]. Nos interesa, entonces, conocer el impacto que puede tener la combinación de criterios en las métricas anteriores.

Es importante notar que los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` no son conflictivos entre sí, según la definición 2.3.1. Dado un conjunto de casos de test con determinadas coberturas para cada uno de estos criterios, el agregado de un nuevo caso de test que aumenta la cobertura de uno de ellos nunca hace disminuir la cobertura de alguno de los otros. Esto ocurre, sencillamente, porque ningún caso de test puede disminuir la cobertura de ninguno de estos criterios, sólo aumentarla: No se puede cubrir menos líneas, ramas, transiciones de EPA, o transiciones de EPA inválidas agregando un nuevo caso de test.

Pregunta 5. *¿Cuán efectivo es el prototipo adaptado, en particular, su uso con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, en generar test suites con buena cobertura de transiciones de EPA?*

Pregunta 6. *¿Cuán efectivo es el prototipo, utilizado con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, para generar casos de test con buena detección de fallas?*

Pregunta 7. *¿Cuán efectivo es el prototipo, utilizado con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, para generar casos de test con buena cobertura de código (líneas y ramas)?*

Pregunta 8. *¿Cuán efectivo es el prototipo, utilizado con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, en detectar transiciones de EPA inválidas?*

4.2. Diseño experimental

Con el objetivo de responder las preguntas enunciadas en 4.1, realizamos una serie de experimentos. Elegimos, como sujetos experimentales, cinco clases de Java. Al elegirlas, se buscó componentes que presentaran restricciones claras en cuanto al orden en que sus operaciones deben ser llamadas y que además contaran con cierta relevancia en la industria por tratarse de componentes de uso común.

Los sujetos elegidos fueron:

[1] Como se menciona en la sección 4.2, la forma en que encaramos nuestra experimentación fue a partir de la alteración de los sujetos (implementaciones de la EPA), y asumimos que la EPA es correcta.

Fig. 4.1: EPA correspondiente a *ListIterator*

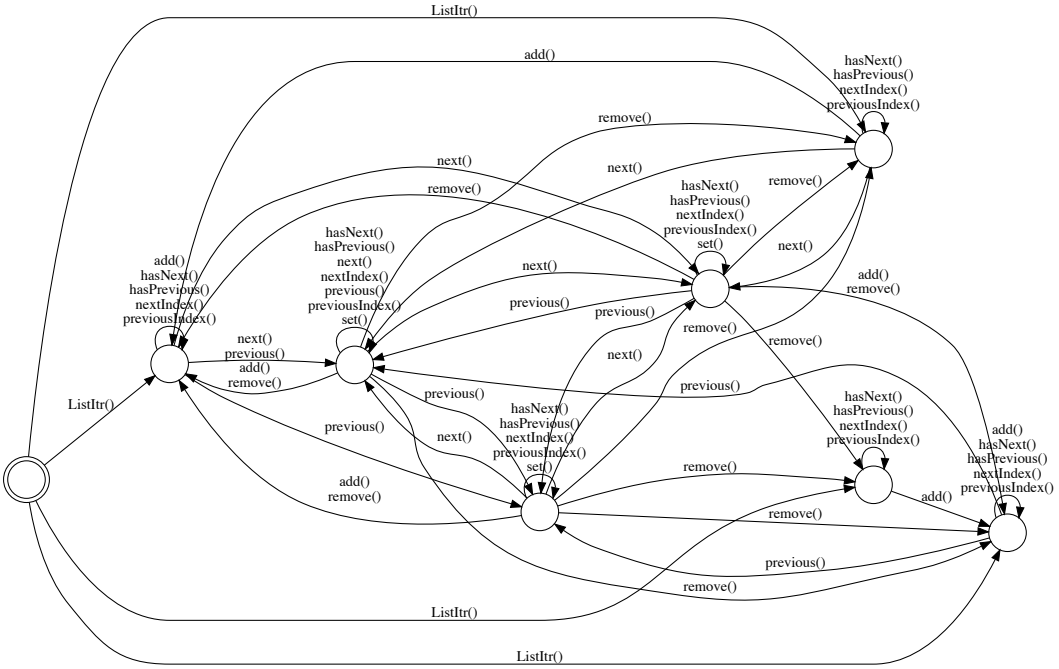


Fig. 4.2: EPA correspondiente a *StringTokenizer*

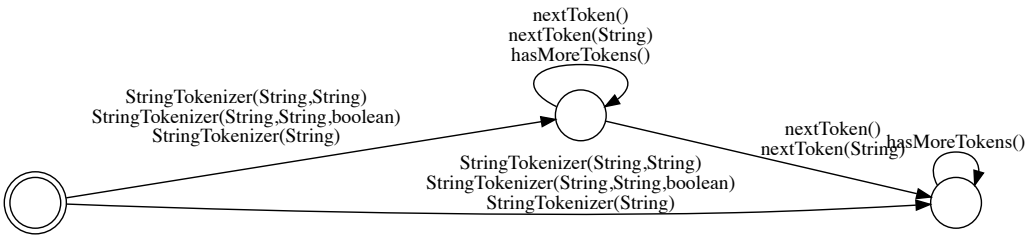


Fig. 4.3: EPA correspondiente a *NumberFormatStringTokenizer*

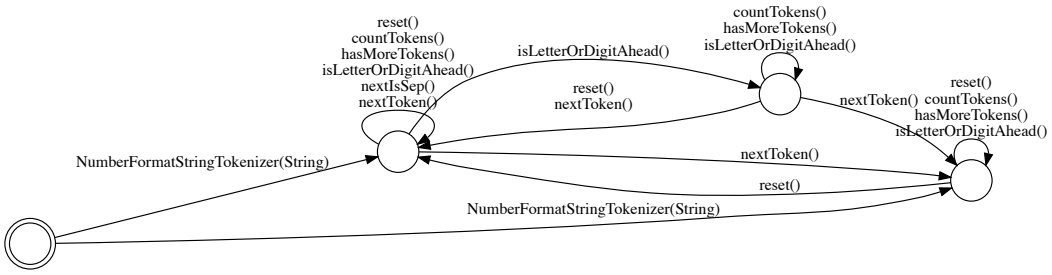
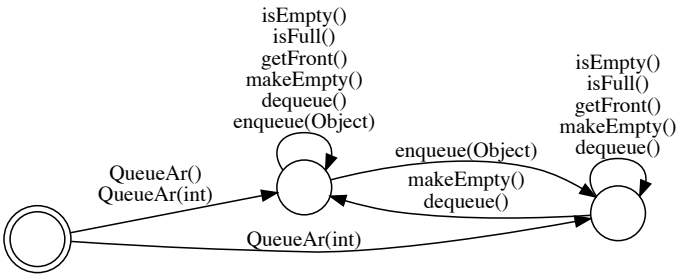


Fig. 4.4: EPA correspondiente a *QueueAr*



- *ListIterator*: Una implementación de la interfaz de Java *ListIterator*, parte del Java Development Kit (JDK) 1.7. Se trata de un iterador para una lista que cuenta con un cursor para desplazarse por la misma, y permite agregar, quitar o modificar elementos en la lista. Una versión anterior de esta misma clase es parte de los sujetos experimentales en [1]. Puede verse una representación de su EPA en la figura 4.1.
- *BoundedStack*: Una pila de objetos sencilla, con una capacidad definida (fijada en 10) y operaciones para apilar y desapilar. Esta estructura está inspirada en `StackAr`, de [19]. Puede verse una representación de su EPA en la figura 3.1.
- *StringTokenizer*: Clase que permite desglosar una cadena de caracteres en componentes (*tokens*), para luego contarlos y obtenerlos desplazándose en la cadena original. Este componente es uno de los sujetos del trabajo [20]. Puede verse una representación de su EPA en la figura 4.2.
- *NumberFormatStringTokenizer*: Componente parte de la biblioteca Apache Xalan. Similar a *StringTokenizer* pero usa como delimitador cualquier caracter no-alfanumérico. Este componente también es uno de los sujetos del trabajo [20]. Puede verse una representación de su EPA en la figura 4.3.
- *QueueAr*: Una cola de objetos, con una capacidad definida y operaciones básicas. La implementación es de [19]. Puede verse una representación de su EPA en la figura 4.4.

La siguiente tabla muestra, para cada sujeto experimental, la cantidad de líneas de código en su código original (sin instrumentación), así como el número de estados, acciones y transiciones en su EPA asociada.

Subject	LOC	Estados de EPA	Acciones de EPA	Transiciones de EPA
ListIterator	610	8	10	69
BoundedStack	21	4	3	7
NFST	64	4	7	21
QueueAr	119	3	8	17
StringTokenizer	156	3	6	12

4.2.1. Comportamiento de EvoSuite con los nuevos criterios

Para responder las preguntas 5 a 7, se llevaron a cabo los siguientes procedimientos:

1. **Generación de casos de test:** Utilizamos herramientas de generación automática de tests para generar un cojunto de casos de test para cada uno de los sujetos experimentales. Las herramientas que utilizamos fueron:

   - EvoSuite con los criterios `LINE`, `BRANCH`
   - EvoSuite con el criterio `EPATRANSITION`
   - EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`

   En cada caso, se usaron límites de tiempo de uno, dos y cinco minutos, y cada experimento se repitió veinte veces.

2. **Evaluación de conjuntos de casos de test:** Para evaluar cada uno de los conjuntos de casos de test generados, utilizamos EvoSuite con su función de medición de cobertura (`-measureCoverage`), que permite obtener la cobertura de líneas, de ramas, *mutation score* y, en nuestro prototipo, de transiciones de EPA válidas e inválidas.

Notar que, en esta sección y en las siguientes, se utiliza a EvoSuite con los criterios `LINE` y `BRANCH` como la herramienta *baseline* contra la cual se compararán el resto de las herramientas. Elegimos esta combinación como baseline por tratarse de una combinación (i) existente en la herramienta original, (ii) estándar en EvoSuite, y (iii) que representa el uso básico y típico de las herramientas de generación de tests: generar conjuntos de casos de test con buena cobertura de código.

4.2.2. Comportamiento del prototipo en la detección de transiciones de EPA inválidas

Las preguntas 8 y 4 apuntan a la evaluación del prototipo como herramienta para generar conjuntos de casos de test que permitan evidenciar transiciones de EPA inválidas. Dado que cada uno de los sujetos experimentales cuenta con una EPA que modela su comportamiento (la que asumimos correcta), se precisa, para poder detectar transiciones inválidas en el código (i) alterar la EPA original, para lo que no contábamos con una forma automática de hacerlo, o bien (ii) versiones alteradas de los sujetos que no respeten la EPA en su totalidad.

Elegimos esta última opción, dado que es posible generar fácil y automáticamente versiones alteradas de los sujetos experimentales mediante algún sistema de mutación. Para generar estas versiones alteradas, utilizamos el sistema muJava, con la configuración que acepta todas las mutaciones disponibles. Para cada sujeto, generamos todos los mutantes con la herramienta muJava, y luego nos quedamos con 50 de estos mutantes elegidos al azar.

Para cada uno de estos sujetos generados, generamos casos de test con las siguientes herramientas:

- EvoSuite con los criterios `LINE`, `BRANCH`
- EvoSuite con el criterio `EPAERROR`
- EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`

4.3. Entorno en el que se realizaron los experimentos

Todos los experimentos fueron realizados en la siguiente configuración de hardware:

- Procesador: Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz (4 cores)
- Memoria: 4 GB

El sistema operativo utilizado fue Ubuntu Linux 16.04 LTS. Se tuvo especial cuidado en reducir la carga del sistema al mínimo al momento de ejecutar los experimentos.

Como parte del trabajo para esta tesis, se desarrolló un sistema que facilitó la especificación de experimentos de generación de conjuntos de casos de test y su evaluación, así

como la ejecución de los mismos. Este sistema ya se encuentra a disposición de los equipos de investigación para ser aprovechado.

4.4. Respuestas

Nomenclatura

En las tablas presentes en esta sección se utiliza la siguiente nomenclatura para distinguir las distintas combinaciones de criterios utilizadas en EvoSuite, y los distintos tipos de métricas.

Herramientas

EvoSuite `ERR` EvoSuite con el criterio `EPAERROR`

EvoSuite `EPA` EvoSuite con el criterio `EPATRANSITION`

EvoSuite `L+B` EvoSuite con los criterios `LINE` y `BRANCH`

EvoSuite `L+B+EPA+ERR` EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`

Métricas

`BRANCH` Cobertura de ramas de código

`EPA` Cobertura de transiciones de EPA

`ERRF` Sujetos inválidos encontrados

`LINE` Cobertura de líneas de código

`MUT` Mutation score

`NERR` Transiciones inválidas encontradas

4.4.1. Criterios simples

Pregunta 1. *¿Cuán efectivo es el prototipo adaptado, utilizado únicamente con el criterio agregado `EPATRANSITION`, en generar test suites con buena cobertura de transiciones de EPA?*

La tabla 4.1 muestra el promedio de la cobertura de transiciones de EPA para cada uno de los sujetos y tiempos límite (*budget*).

El **p-valor** es una medida estadística estándar que, en este caso, representa la probabilidad de que, dada la hipótesis nula de que no hay diferencias entre los algoritmos en cuestión (en este caso, EvoSuite con `LINE` y `BRANCH`, por un lado, y EvoSuite con `EPATRANSITION`, por el otro) en nuestra experimentación, esta sea rechazada siendo cierta [21]. Para establecer un límite entre los resultados estadísticamente significativos y los que no lo son, debe elegirse un nivel de significación α, en otras palabras, un valor máximo de p-valor a partir del cual ya no podemos rechazar la hipótesis nula, es decir, no podemos decir que hay diferencias entre los algoritmos. Elegimos $\alpha = 0{,}05$, por ser lo aceptado empíricamente en la bibliografía.

Tab. 4.1: Resultados al ejecutar EvoSuite con los los criterios `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y `EPATRANSITION` (EvoSuite `EPA`) por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` EPA	EvoSuite `EPA` EPA	p-valor	A_{12}
ListIterator	60	0.24	0.94	**0.00000001**	**0.00000000**
	120	0.25	0.94	**0.00000001**	**0.00000000**
	300	0.23	0.94	**0.00000001**	**0.00000000**
BoundedStack	60	0.71	0.99	**0.00000000**	**0.02500000**
	120	0.71	1.00	**0.00000000**	**0.00000000**
	300	0.71	1.00	**0.00000000**	**0.00000000**
NFST	60	0.51	0.67	**0.00000001**	**0.00000000**
	120	0.53	0.67	**0.00000000**	**0.00000000**
	300	0.54	0.67	**0.00000000**	**0.00000000**
QueueAr	60	0.71	1.00	**0.00000001**	**0.00000000**
	120	0.71	1.00	**0.00000001**	**0.00000000**
	300	0.72	1.00	**0.00000000**	**0.00000000**
StringTokenizer	60	0.61	0.87	**0.00000008**	**0.01500000**
	120	0.62	0.86	**0.00000008**	**0.01625000**
	300	0.64	0.85	**0.00000006**	**0.01875000**

La medida A_{12} es el ***effect size*** descripto por Vargha y Delaney en [22], y representa la probabilidad de que el algoritmo de la izquierda permita obtener valores más altos que el algoritmo de la derecha [21] para nuestro experimento. En otras palabras, si el valor de A_{12} es cercano a 0, esto significa que el algoritmo de la derecha obtiene valores más altos, mientras que si es cercano a 1, eso significa que el algoritmo de la izquierda es el que obtiene los valores más altos. Un valor de 0.5 indica que no hay diferencia entre los algoritmos.

La tabla 4.1 deja ver que, para todos los sujetos experimentales, el p-valor obtenido a partir de los resultados se encuentra por debajo de nuestro nivel de significación, y el valor de A_{12} es, en todos los casos cercano a 0.

> El uso de EvoSuite con el criterio `EPATRANSITION` permite obtener una mayor cobertura de transiciones de EPA que EvoSuite usado sólo con `LINE` y `BRANCH`, en todos los casos observados.

Pregunta 2. *¿Cuán efectivo es el prototipo, utilizado únicamente con el criterio agregado `EPATRANSITION` para generar casos de test con buena detección de fallas?*

Una forma de medir la efectividad de un conjunto de casos de test en la detección de fallas es el puntaje conocido como **"mutation score"**. Esta medida surge del ámbito de *mutation testing*. La idea aquí es tomar el código que pasa correctamente los casos de test y efectuarle modificaciones (mutaciones) utilizando operadores que imitan errores comunes en programación. Si un conjunto de casos de test falla al ejecutarse con el código mutado ("mata al mutante") y otro no, esto sirve de muestra de que el primer conjunto detecta un tipo específico de falla que el segundo no. El mutation score representa el porcentaje de mutantes (del total de mutante generados) que fueron detectados (matados) por un conjunto de casos de test [23].

Existen, además, dos tipos de mutation testing: El débil, y el fuerte. El mutation testing débil requiere, para poder considerar a un mutante matado, que el conjunto de casos de

Tab. 4.2: Mutation score al ejecutar EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y EvoSuite con `EPATRANSITION` (EvoSuite `EPA`) por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` MUT	EvoSuite `EPA` MUT	p-valor	A_{12}
ListIterator	60	0.92	0.87	**0.00000000**	**1.00000000**
	120	0.92	0.87	**0.00000000**	**1.00000000**
	300	0.92	0.87	**0.00000000**	**1.00000000**
BoundedStack	60	0.96	0.87	**0.00000000**	**1.00000000**
	120	0.96	0.87	**0.00000000**	**1.00000000**
	300	0.96	0.87	**0.00000000**	**1.00000000**
NFST	60	0.94	0.90	**0.00000008**	**0.95000000**
	120	0.94	0.88	**0.00000003**	**0.97500000**
	300	0.94	0.87	**0.00000001**	**1.00000000**
QueueAr	60	0.94	0.91	**0.00000497**	**0.85000000**
	120	0.94	0.92	**0.00012882**	**0.77500000**
	300	0.94	0.92	**0.00033383**	**0.75000000**
StringTokenizer	60	0.64	0.49	**0.00000003**	**1.00000000**
	120	0.64	0.50	**0.00000003**	**1.00000000**
	300	0.64	0.49	**0.00000002**	**1.00000000**

test ejecute la línea mutada y que, como consecuencia, el estado del objeto sea alterado. El fuerte requiere, además de lo anterior, que alguna aserción detecte la diferencia en el comportamiento. El costo computacional de la variante débil suele ser mucho menor, y experimentos en la bibliografía han mostrado que los resultados de mutation testing débil son casi tan buenos como los de la variante fuerte [24].

Para poder responder la pregunta 2, realizamos una comparación con EvoSuite con los criterios `LINE` y `BRANCH`, utilizando la variante débil de mutation testing. La tabla 4.2 muestra los resultados comparados. Allí se observa que los p-valores están siempre por debajo de nuestro nivel de significación y que los valores de A_{12} son siempre cercanos a 1, indicando que EvoSuite con `LINE` y `BRANCH` siempre obtiene resultados mayores que EvoSuite con `EPATRANSITION` en nuestra experimentación.

> EvoSuite utilizado únicamente con el crtierio `EPATRANSITION` no es más efectivo que EvoSuite con `LINE` y `BRANCH` en cuanto a la capacidad de generar tests con alta detección de fallas en nuestros experimentos.

Pregunta 3. *¿Cuán efectivo es el prototipo, utilizado únicamente con el criterio agregado `EPATRANSITION` para generar casos de test con buena cobertura de código (líneas y ramas)?*

Tab. 4.3: Cobertura de código al ejecutar EvoSuite con los los criterios `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y `EPATRANSITION` (EvoSuite `EPA`) por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` LINE	EvoSuite `L+B` BRNCH	EvoSuite `EPA` LINE	EvoSuite `EPA` BRNCH	p-valor	A_{12}	p-valor	A_{12}
ListIterator	60	0.88	0.89	0.81	0.68	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
	120	0.88	0.89	0.81	0.68	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
	300	0.88	0.89	0.81	0.68	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
BoundedStack	60	1.00	1.00	0.82	0.60	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
	120	1.00	1.00	0.82	0.60	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
	300	1.00	1.00	0.82	0.60	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
NFST	60	1.00	0.93	0.92	0.82	**0.00000001**	**1.00000000**	**0.00000001**	**1.00000000**
	120	1.00	0.93	0.91	0.79	**0.00000001**	**1.00000000**	**0.00000001**	**1.00000000**
	300	1.00	0.93	0.91	0.76	**0.00000001**	**1.00000000**	**0.00000000**	**1.00000000**
QueueAr	60	1.00	1.00	0.95	0.91	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
	120	1.00	1.00	0.95	0.91	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
	300	1.00	1.00	0.96	0.92	**0.00000000**	**1.00000000**	**0.00000000**	**1.00000000**
StringTokenizer	60	0.77	0.62	0.57	0.43	**0.00000001**	**1.00000000**	**0.00000001**	**1.00000000**
	120	0.77	0.62	0.57	0.43	**0.00000001**	**1.00000000**	**0.00000001**	**1.00000000**
	300	0.77	0.62	0.56	0.42	**0.00000001**	**1.00000000**	**0.00000001**	**1.00000000**

Tab. 4.4: Cantidad promedio de transiciones inválidas en los mutantes generados (50 por cada sujeto) encontradas al ejecutar EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y EvoSuite con `EPAERROR` (EvoSuite `ERR`) por el otro, con 5 repeticiones. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` NERR	EvoSuite `ERR` NERR	p-valor	A_{12}
ListIterator	60	10.60	34.40	**0.01090950**	**0.00000000**
BoundedStack	60	2.00	4.00	**0.00397675**	**0.00000000**
NFST	60	2.00	9.80	**0.00923673**	**0.00000000**
QueueAr	60	2.80	10.00	**0.00708872**	**0.00000000**
StringTokenizer	60	0.60	4.00	**0.00650170**	**0.00000000**

La tabla 4.3 muestra las coberturas de líneas y ramas obtenidas utilizando EvoSuite con los criterios `LINE` y `BRANCH` en comparación con las obtenidas por EvoSuite utilizando únicamente `EPATRANSITION`. Los p-valores están siempre por debajo de nuestro nivel de significación y el valor de A_{12} es siempre 1.

> El uso de EvoSuite únicamente con el criterio `EPATRANSITION` es menos efectivo que su uso con los criterios `LINE` y `BRANCH` en la obtención de mayores coberturas de código en los casos analizados.

Pregunta 4. *¿Cuán efectivo es el prototipo, utilizado únicamente con el criterio agregado `EPAERROR` en detectar transiciones de EPA inválidas?*

Las tablas 4.4 y 4.5 muestran las cantidades promedio de transiciones y mutantes inválidos detectados, en el conjunto de mutantes generados (50 por cada sujeto) al ejecutar EvoSuite con `LINE` y `BRANCH`, por un lado, y EvoSuite con `EPAERROR` por el otro, con 5 repeticiones. Los p-valores están siempre por debajo de nuestro nivel de significación y el valor de A_{12} es siempre cercano al cero.

> El prototipo, usado únicamente con el criterio `EPAERROR` es más efectivo que EvoSuite utilizado con `LINE` y `BRANCH` a la hora de detectar transiciones inválidas y sujetos inválidos en nuestros experimentos.

4.4.2. Criterios combinados

Vale la pena recordar que nos interesa examinar los resultados con criterios combinados porque existe investigación previa que muestra que coberturas individuales pueden ser potenciadas mediante la combinación de criterios no conflictivos entre sí [16].

Pregunta 5. *¿Cuán efectivo es el prototipo adaptado, en particular, su uso con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, en generar test suites con buena cobertura de transiciones de EPA?*

La tabla 4.6 es análoga a la tabla 4.1 sólo que la comparación es contra EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`, en lugar de únicamente `EPATRANSITION`. Los p-valores están siempre por debajo de nuestro nivel de significación y el valor de A_{12} es siempre cercano a 0.

Tab. 4.5: Cantidad promedio de mutantes inválidos detectados del total de los mutantes generados (50 por cada sujeto) al ejecutar EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y EvoSuite con `EPAERROR` (EvoSuite `ERR`) por el otro, con 5 repeticiones. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` ERRF	EvoSuite `ERR` ERRF	p-valor	A_{12}
ListIterator	60	6.80	8.00	**0.00558362**	**0.00000000**
BoundedStack	60	2.00	4.00	**0.00397675**	**0.00000000**
NFST	60	2.00	6.80	**0.00923673**	**0.00000000**
QueueAr	60	2.40	4.00	**0.00650170**	**0.00000000**
StringTokenizer	60	0.60	2.00	**0.00650170**	**0.00000000**

Tab. 4.6: Resultados al ejecutar EvoSuite con los los criterios `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`) por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` EPA	EvoSuite `L+B+EPA+ERR` EPA	p-valor	A_{12}
ListIterator	60	0.24	0.84	**0.00000006**	**0.00000000**
	120	0.25	0.87	**0.00000006**	**0.00000000**
	300	0.23	0.89	**0.00000005**	**0.00000000**
BoundedStack	60	0.71	1.00	**0.00000000**	**0.00000000**
	120	0.71	1.00	**0.00000000**	**0.00000000**
	300	0.71	1.00	**0.00000000**	**0.00000000**
NFST	60	0.51	0.66	**0.00000009**	**0.03875000**
	120	0.53	0.66	**0.00000004**	**0.02875000**
	300	0.54	0.67	**0.00000000**	**0.00000000**
QueueAr	60	0.71	1.00	**0.00000001**	**0.00000000**
	120	0.71	1.00	**0.00000001**	**0.00000000**
	300	0.72	1.00	**0.00000000**	**0.00000000**
StringTokenizer	60	0.61	0.89	**0.00000006**	**0.00875000**
	120	0.62	0.87	**0.00000008**	**0.01500000**
	300	0.64	0.88	**0.00000008**	**0.01375000**

El prototipo (esta vez con criterios adicionales) permite obtener una mayor cobertura de transiciones de EPA que EvoSuite usado sólo con `LINE` y `BRANCH`, en todos los casos observados.

Pregunta 6. *¿Cuán efectivo es el prototipo, utilizado con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, para generar casos de test con buena detección de fallas?*

La tabla 4.7 nos muestra que, para el caso de *ListIterator*, la herramienta utilizada con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`, permite obtener un mayor *mutation score* (p-valores y A_{12} cercanos a cero). En cuanto al resto de los sujetos, se observa, o bien que los resultados no son significativos (p-valor por encima de nuestro nivel de significación), o bien que los resultados para ambas herramientas son idénticos (p-valor etiquetado como "nan").

Para ahondar más en estos resultados, realizamos un análisis de mutación fuerte (*strong mutation*). La mutación fuerte, como dijimos, es computacionalmente más cara, pero, en algunos casos, puede dar mejores resultados. Los resultados de este análisis pueden verse en la tabla 4.8. Estos resultados confirman la observación anterior en cuanto a *ListIterator*,

Tab. 4.7: Mutation score al ejecutar EvoSuite con los los criterios `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`) por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` MUT	EvoSuite `L+B+EPA+ERR` MUT	p-valor	A_{12}
ListIterator	60	0.92	0.94	**0.00000000**	**0.00000000**
	120	0.92	0.94	**0.00000000**	**0.00000000**
	300	0.92	0.94	**0.00000000**	**0.02500000**
BoundedStack	60	0.96	0.96	nan	0.50000000
	120	0.96	0.96	nan	0.50000000
	300	0.96	0.96	nan	0.50000000
NFST	60	0.94	0.94	nan	0.50000000
	120	0.94	0.94	0.34211230	0.52500000
	300	0.94	0.94	nan	0.50000000
QueueAr	60	0.94	0.94	nan	0.50000000
	120	0.94	0.94	nan	0.50000000
	300	0.94	0.94	nan	0.50000000
StringTokenizer	60	0.64	0.64	0.75149650	0.52500000
	120	0.64	0.64	1.00000000	0.50000000
	300	0.64	0.64	0.50646600	0.55000000

además de mostrar que aplica también para *BoundedStack*. El resto de los resultados son mayormente no significativos (p-valor muy alto) pero, cuando lo son, se inclinan hacia el criterio combinado de `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` como más efectivo a la hora de obtener un mayor *mutation score*.

> Los resultados muestran que el combinado de `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` es más efectivo a la hora de obtener un mayor *mutation score* en comparación con EvoSuite con `LINE` y `BRANCH` en una parte de los casos evaluados, mientras que los resultados para el resto de los casos no son concluyentes.

Pregunta 7. *¿Cuán efectivo es el prototipo, utilizado con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, para generar casos de test con buena cobertura de código (líneas y ramas)?*

Tab. 4.8: Mutation score (fuerte) al ejecutar EvoSuite con los los criterios `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`) por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` MUT	EvoSuite `L+B+EPA+ERR` MUT	p-valor	A_{12}
ListIterator	60	0.70	0.81	**0.00000005**	**0.00000000**
	120	0.71	0.81	**0.00000004**	**0.00000000**
	300	0.70	0.81	**0.00000005**	**0.00000000**
BoundedStack	60	0.78	0.85	**0.00000001**	**0.00000000**
	120	0.78	0.86	**0.00000000**	**0.00000000**
	300	0.78	0.88	**0.00000002**	**0.02875000**
NFST	60	0.71	0.70	0.75443460	0.53000000
	120	0.70	0.69	0.54931120	0.55625000
	300	0.70	0.71	0.06741441	0.33125000
QueueAr	60	0.64	0.67	**0.00003503**	**0.12500000**
	120	0.65	0.67	0.14002560	0.36500000
	300	0.65	0.70	**0.00021150**	**0.16250000**
StringTokenizer	60	0.40	0.40	0.62184660	0.54625000
	120	0.40	0.39	0.25413280	0.60500000
	300	0.38	0.39	0.41388340	0.42375000

Tab. 4.9: Cobertura de código al ejecutar EvoSuite con los los criterios `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y EvoSuite con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`) por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` LINE	EvoSuite `L+B` BRNCH	EvoSuite `L+B+EPA+ERR` LINE	EvoSuite `L+B+EPA+ERR` BRNCH	p-valor	A_{12}	p-valor	A_{12}
ListIterator	60	0.88	0.89	0.88	0.89	nan	0.50000000	nan	0.50000000
	120	0.88	0.89	0.88	0.89	nan	0.50000000	nan	0.50000000
	300	0.88	0.89	0.88	0.89	0.34211230	0.52500000	0.34211230	0.52500000
BoundedStack	60	1.00	1.00	1.00	1.00	nan	0.50000000	nan	0.50000000
	120	1.00	1.00	1.00	1.00	nan	0.50000000	nan	0.50000000
	300	1.00	1.00	1.00	1.00	nan	0.50000000	nan	0.50000000
NFST	60	1.00	0.93	1.00	0.93	0.34211230	0.52500000	0.34211230	0.52500000
	120	1.00	0.93	0.99	0.92	0.34211230	0.52500000	0.34211230	0.52500000
	300	1.00	0.93	1.00	0.93	nan	0.50000000	nan	0.50000000
QueueAr	60	1.00	1.00	1.00	1.00	nan	0.50000000	nan	0.50000000
	120	1.00	1.00	1.00	1.00	nan	0.50000000	nan	0.50000000
	300	1.00	1.00	1.00	1.00	nan	0.50000000	nan	0.50000000
StringTokenizer	60	0.77	0.62	0.77	0.62	nan	0.50000000	0.34211230	0.52500000
	120	0.77	0.62	0.77	0.62	nan	0.50000000	nan	0.50000000
	300	0.77	0.62	0.77	0.62	nan	0.50000000	nan	0.50000000

Tab. 4.10: Cantidad promedio de transiciones inválidas en los mutantes generados (50 por cada sujeto) encontradas al ejecutar EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y EvoSuite con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`) por el otro, con 5 repeticiones. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` NERR	EvoSuite `L+B+EPA+ERR` NERR	p-valor	A_{12}
ListIterator	60	10.60	30.40	**0.00993701**	**0.00000000**
BoundedStack	60	2.00	4.00	**0.00397675**	**0.00000000**
NFST	60	2.00	8.00	**0.01141204**	**0.00000000**
QueueAr	60	2.80	8.00	**0.01501859**	**0.02000000**
StringTokenizer	60	0.60	4.00	**0.00650170**	**0.00000000**

La tabla 4.9 compara la cobertura de código entre EvoSuite utilizado con `LINE` y `BRANCH`, y EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`. En esta tabla, algunos p-valores están por encima del nivel de significación y, por lo tanto, sus resultados asociados no pueden ser considerados significativos, mientras que el resto están marcados con la etiqueta "nan", la que indica que no se observó diferencias entre los resultados de uno u otro algoritmo, incluso luego de las 20 repeticiones del experimento.

> Los resultados obtenidos no permiten concluir que EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` sea igual, más o menos efectivo en la cobertura de código que EvoSuite utilizado con `LINE` y `BRANCH`, aunque sugieren que no hay diferencias significativas entre ambas herramientas en lo que a cobertura de código se refiere para los casos analizados.

Pregunta 8. *¿Cuán efectivo es el prototipo, utilizado con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en conjunto, en detectar transiciones de EPA inválidas?*

Las tablas 4.10 y 4.11 son análogas a las tablas 4.4 y 4.5, sólo que utilizando el criterio combinado con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` en lugar de únicamente `EPATRANSITION`. La mayoría de los p-valores están por debajo del nivel de significación y los valores de A_{12} son siempre cercanos a cero.

> El prototipo, con el criterio combinado, es más efectivo que EvoSuite utilizado con `LINE` y `BRANCH` a la hora de detectar transiciones inválidas y sujetos inválidos en todos los casos observados.

4.4.3. Comparación entre herramientas con mejores resultados

Las respuestas a las preguntas 1 y 5, por un lado, y a las preguntas 4 y 8, por el otro, muestran que el prototipo que construimos es más efectivo para conseguir mayores coberturas de EPA y para detectar transiciones y sujetos que no cumplen con la especificación de la EPA, en comparación con la herramienta original, para los casos observados. Nos interesa comparar entre sí a las variantes que obtuvieron mejores resultados que la versión original de EvoSuite.

Tab. 4.11: Cantidad promedio de de mutantes inválidos detectados del total de los mutantes generados (50 por cada sujeto) al ejecutar EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`), por un lado, y EvoSuite con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`) por el otro, con 5 repeticiones. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B` ERRF	EvoSuite `L+B+EPA+ERR` ERRF	p-valor	A_{12}
ListIterator	60	6.80	8.60	**0.00856192**	**0.00000000**
BoundedStack	60	2.00	4.00	**0.00397675**	**0.00000000**
NFST	60	2.00	5.60	**0.01115943**	**0.00000000**
QueueAr	60	2.40	3.20	0.05582929	0.16000000
StringTokenizer	60	0.60	2.00	**0.00650170**	**0.00000000**

Tab. 4.12: Resultados al ejecutar EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`), por un lado, y únicamente con `EPATRANSITION` (EvoSuite `EPA`), por el otro. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B+EPA+ERR` EPA	EvoSuite `EPA` EPA	p-valor	A_{12}
ListIterator	60	0.84	0.94	**0.00000001**	**0.00000000**
	120	0.87	0.94	**0.00000001**	**0.00125000**
	300	0.89	0.94	**0.00000006**	**0.03250000**
BoundedStack	60	1.00	0.99	0.34211230	0.52500000
	120	1.00	1.00	nan	0.50000000
	300	1.00	1.00	nan	0.50000000
NFST	60	0.66	0.67	0.34211230	0.47500000
	120	0.66	0.67	0.34211230	0.47500000
	300	0.67	0.67	nan	0.50000000
QueueAr	60	1.00	1.00	nan	0.50000000
	120	1.00	1.00	nan	0.50000000
	300	1.00	1.00	nan	0.50000000
StringTokenizer	60	0.89	0.87	0.09872921	0.63500000
	120	0.87	0.86	0.59935820	0.54250000
	300	0.88	0.85	0.12987520	0.61875000

La tabla 4.12 muestra que, para el caso de *ListIterator*, el prototipo utilizado únicamente con el criterio `EPATRANSITION` obtiene mayores coberturas de EPA que el prototipo usando el criterio combinado (p-valor debajo del nivel de significación y A_{12} cercano a cero). Sin embargo, para el resto de los sujetos experimentales, incluso luego de 20 repeticiones, no se pudo obtener datos que permitan inclinarse para uno u otro prototipo; en algunos casos el p-valor es alto, y en otros los resultados fueron idénticos para ambos algoritmos.

> Los resultados obtenidos no nos permiten inclinarnos hacia una u otra combinación de criterios a la hora de buscar una mayor cobertura de EPA, aunque se observó que el prototipo con el criterio `EPATRANSITION` fue más efectivo que con el criterio combinado para uno de los sujetos experimentales.

En cuanto a la detección de transiciones inválidas, la tabla 4.13 muestra que el uso del prototipo utilizando únicamente `EPAERROR` se mostró más efectivo para *QueueAr* (p-valores por debajo del nivel de significación y A_{12} cercano a cero). Para el resto de los

Tab. 4.13: Cantidad promedio de transiciones inválidas en los mutantes generados (50 por cada sujeto) encontradas al ejecutar EvoSuite con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`), por un lado, y EvoSuite con `EPAERROR` por el otro, con 5 repeticiones. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B+EPA+ERR` NERR	EvoSuite `ERR` NERR	p-valor	A_{12}
ListIterator	60	30.40	34.40	0.09068837	0.16000000
BoundedStack	60	4.00	4.00	nan	0.50000000
NFST	60	8.00	9.80	0.19381570	0.26000000
QueueAr	60	8.00	10.00	**0.02480842**	**0.10000000**
StringTokenizer	60	4.00	4.00	nan	0.50000000

Tab. 4.14: Cantidad promedio de de mutantes inválidos detectados del total de los mutantes generados (50 por cada sujeto) al ejecutar EvoSuite con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` (EvoSuite `L+B+EPA+ERR`), por un lado, y EvoSuite con `EPAERROR` por el otro, con 5 repeticiones. Los resultados estadísticamente significativos se muestran en negrita.

Subject	Budget	EvoSuite `L+B+EPA+ERR` ERRF	EvoSuite `ERR` ERRF	p-valor	A_{12}
ListIterator	60	8.60	8.00	0.06679807	0.80000000
BoundedStack	60	4.00	4.00	nan	0.50000000
NFST	60	5.60	6.80	0.19208380	0.26000000
QueueAr	60	3.20	4.00	**0.01996445**	**0.10000000**
StringTokenizer	60	2.00	2.00	nan	0.50000000

sujetos los resultados no permiten inclinarse hacia ninguno de los dos prototipos como el más efectivo; como en la tabla anterior, en algunos casos el p-valor es alto, y en otros los resultados se repitieron para ambos algoritmos.

La tabla 4.14 muestra la comparación en cuanto a la detección de sujetos inválidos. En este caso, vuelve a observarse que, para *QueueAr*, el prototipo utilizado únicamente con `EPAERROR` es más efectivo. No hay información para inclinarse por una herramienta u otra para el resto de los sujetos, nuevamente a causa de p-valores altos o resultados idénticos entre ambas herramientas.

> Los resultados obtenidos no nos permiten inclinarnos definitivamente hacia una u otra combinación de criterios a la hora de elegir la más efectiva para la detección de transiciones y sujetos inválidos.

4.4.4. Correlación entre coberturas

Como mencionamos anteriormente, existe investigación previa que sugiere que la alta cobertura de la EPA de un componente, por parte de una test suite, es un buen predictor del número de fallas que el test suite puede llegar a encontrar, así como de la cobertura de código (líneas y ramas) que puede alcanzar[1]. Debido a esto, esperábamos observar, en nuestra experimentación, que, en los casos en los cuales una herramienta se mostraba superior a otra para obtener coberturas de EPA, se mostrara también superior al momento de conseguir mayor detección de fallas (mutation score) y mayores coberturas de código

Tab. 4.15: Correlaciones según el coeficiente de Pearson entre cobertura de transiciones de EPA y mutation score para los conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`) y EvoSuite con `EPATRANSITION` (EvoSuite `EPA`).

Subject	Budget	p-valor	Corr EPA,MUT
ListIterator	60	**0.00000000**	**-0.99053850**
	120	**0.00000000**	**-0.99347140**
	300	**0.00000000**	**-0.99348150**
BoundedStack	60	**0.00000000**	**-0.95118970**
	120	**0.00000000**	**-1.00000000**
	300	**0.00000000**	**-1.00000000**
NFST	60	**0.00028172**	**-0.54440800**
	120	**0.00000536**	**-0.65118440**
	300	**0.00000001**	**-0.77031790**
QueueAr	60	**0.00000015**	**-0.72112020**
	120	**0.00004333**	**-0.59963170**
	300	**0.00012934**	**-0.56857800**
StringTokenizer	60	**0.00000000**	**-0.83909060**
	120	**0.00000000**	**-0.78684700**
	300	**0.00000000**	**-0.80732410**

(líneas y ramas). Esto no fue así: EvoSuite usado con `EPATRANSITION` se mostró más efectivo en conseguir coberturas de EPA que EvoSuite con `LINE` y `BRANCH` y, sin embargo, se mostró menos efectivo en la obtención de mayores coberturas de código y detección de fallas. El caso del prototipo con el criterio combinado (`LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`), que también se mostró más efectivo en la obtención de mayores coberturas de EPA, fue menos concluyente, pero tampoco corroboró nuestra hipótesis.

Decidimos realizar un análisis de correlación entre coberturas para intentar exponer, mediante un mecanismo estándar, estas observaciones. Para esto, usamos el coeficiente de correlación de Pearson, ampliamente utilizado en la bibliografía. El coeficiente de Pearson es un número en el rango $[-1, 1]$ que puede interpretarse como un indicador de la correlación lineal entre dos variables [25]. Un valor de 1 indica que una ecuación lineal describe perfectamente la relación entre ambas variables de manera tal que un incremento en el valor de una de las variables implica un incremento en la otra. Un valor de -1 indica que los valores de ambas variables podrían representarse en una recta de forma tal que el valor de una variable decrece a medida que aumenta el valor de la otra. El p-valor representa la probabilidad de que, dada la hipótesis nula de que la correlación es igual a cero, esta sea rechazada siendo cierta. Nuevamente tomamos $\alpha = 0{,}05$ como nivel de significación.

La tabla 4.15 muestra el coeficiente de correlación de Pearson para la cobertura de transiciones de EPA y mutation score débil, para conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH` y EvoSuite con `EPATRANSITION`. Esta tabla expresa que existe una correlación negativa entre estas dos métricas, para los conjuntos de casos de test generados con estas herramientas. Esto se condice con lo observado previamente, en las tablas correspondientes a las respuestas a las preguntas 1 y 2.

La tabla 4.16 es análoga a la 4.15, pero tomando los resultados para los conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH`, y con EvoSuite con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`. Aquí también se corrobora lo observado en las tablas anteriores, en este caso las correspondientes a las respuestas a las preguntas 5 y 6. La tabla 4.17 es la versión de la tabla 4.16 para mutación fuerte (strong mutation). Estas tablas muestran que existe una correlación positiva entre cobertura de transiciones de EPA y

Tab. 4.16: Correlaciones según el coeficiente de Pearson entre cobertura de transiciones de EPA y mutation score (débil) para los conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`) y EvoSuite con `LINE`, `BRANCH`, `EPAERROR` y `EPATRANSITION` (EvoSuite `L+B+EPA+ERR`).

Subject	Budget	p-valor	Corr EPA,MUT
ListIterator	60	**0.00000000**	**0.99559580**
	120	**0.00000000**	**0.99512820**
	300	**0.00000000**	**0.94444850**
BoundedStack	60	nan	nan
	120	nan	nan
	300	nan	nan
NFST	60	nan	nan
	120	0.31075540	0.16438940
	300	nan	nan
QueueAr	60	nan	nan
	120	nan	nan
	300	nan	nan
StringTokenizer	60	0.98954690	0.00213936
	120	0.68727780	-0.06566245
	300	0.31410930	-0.16326930

mutation score, débil y fuerte, para algunos de los sujetos.

La tabla 4.18 muestra las correlaciones entre cobertura de transiciones de EPA y cobertura de código (líneas y ramas) para los conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH`, y EvoSuite con `EPATRANSITION`. Aquí puede observarse una correlación negativa en todos los casos. La tabla 4.19 es análoga pero los conjuntos de casos de test son los generados con EvoSuite con `LINE` y `BRANCH`, y EvoSuite con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`. De esta tabla no pueden sacarse conclusiones, ya que los p-valores son altos, o bien los valores de cobertura de transiciones de EPA son idénticos, lo cual no permite realizar el análisis de correlación (situación representada por el valor "nan").

> Los resultados obtenidos muestran que la correlación entre cobertura de EPA y detección de fallas (mutation score) dependerá de las herramientas utilizadas para generar los conjuntos de casos de test.

4.4.5. Conclusiones de los resultados experimentales

A partir de los resultados anteriores enunciamos las siguientes conclusiones:

- La herramienta extendida, tanto en sus usos con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`, como con `EPATRANSITION` únicamente, se mostraron superiores a EvoSuite con los criterios de `LINE` y `BRANCH` a la hora de lograr **mayores coberturas de EPA**
- La herramienta extendida, tanto en sus usos con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR`, como con `EPAERROR` únicamente, se mostraron superiores a EvoSuite con los criterios de `LINE` y `BRANCH` para **detectar transiciones y sujetos inválidos** (comportamientos que difieren de lo especificado en la EPA)

Tab. 4.17: Correlaciones según el coeficiente de Pearson entre cobertura de transiciones de EPA y mutation score (fuerte) para los conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`) y EvoSuite con `LINE`, `BRANCH`, `EPAERROR` y `EPATRANSITION` (EvoSuite `L+B+EPA+ERR`).

Subject	Budget	p-valor	Corr EPA,MUT
ListIterator	60	**0.00000000**	**0.97342930**
	120	**0.00000000**	**0.97253380**
	300	**0.00000000**	**0.97104780**
BoundedStack	60	**0.00000014**	**0.72208770**
	120	**0.00000000**	**0.78833710**
	300	**0.00000000**	**0.84912630**
NFST	60	0.44732490	-0.12360560
	120	0.74620040	-0.05281309
	300	0.12286010	0.24798590
QueueAr	60	**0.00001514**	**0.62673520**
	120	0.09235904	0.26971410
	300	**0.00016331**	**0.56153000**
StringTokenizer	60	0.58476070	-0.08905227
	120	0.91181260	-0.01808352
	300	0.52065150	0.10460080

- La herramienta extendida, en su uso únicamente con el criterio `EPATRANSITION` se mostró menos efectiva que su uso con los criterios `LINE` y `BRANCH` en la obtención de mayores coberturas de código
- No podemos asegurar que, para todos los sujetos estudiados, la herramienta extendida, en su uso con `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` sea significativamente distinguible a EvoSuite con `LINE` y `BRANCH` en cuanto a la obtención de mayores coberturas de código o detección de fallas

Por lo tanto, deducimos que el uso de EvoSuite con los criterios `LINE`, `BRANCH`, `EPATRANSITION` y `EPAERROR` puede resultar el más conveniente para conseguir mayor coberturas de EPA sin relegar cobertura de código y detección de fallas en comparación con la herramienta original con sus criterios `LINE` y `BRANCH`.

Por último, encontramos que la correlación entre cobertura de EPA y detección de fallas (mutation score) depende fuertemente de las herramientas utilizadas para generar los conjuntos de casos de test.

4.4.6. Amenazas a la validez

Amenazas a la validez externa

Las amenazas a la validez externa son aquellas que ponen en riesgo la generalización de los resultados obtenidos.

Amenaza 1. *Los sujetos experimentales no son representativos de todas las clases que se encuentran en la práctica*

Para mitigar esta amenaza, elegimos sujetos experimentales provenientes de distintos trabajos de la bibliografía. Además, algunos de nuestros sujetos son estructuras de datos ampliamente utilizadas en la industria.

Tab. 4.18: Correlaciones según el coeficiente de Pearson entre cobertura de transiciones de EPA y cobertura de código (líneas y ramas) para los conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`) y EvoSuite con `EPATRANSITION` (EvoSuite `EPA`).

Subject	Budget	p-valor	Corr EPA,LINE	p-valor	Corr EPA,BRNCH
ListIterator	60	**0.00000000**	**-0.99929000**	**0.00000000**	**-0.99929000**
	120	**0.00000000**	**-0.99927720**	**0.00000000**	**-0.99927720**
	300	**0.00000000**	**-0.99956080**	**0.00000000**	**-0.99956080**
BoundedStack	60	**0.00000000**	**-0.95118970**	**0.00000000**	**-0.95118970**
	120	**0.00000000**	**-1.00000000**	**0.00000000**	**-1.00000000**
	300	**0.00000000**	**-1.00000000**	**0.00000000**	**-1.00000000**
NFST	60	**0.00000000**	**-0.85578830**	**0.00000000**	**-0.83049740**
	120	**0.00000000**	**-0.81677780**	**0.00000001**	**-0.77308790**
	300	**0.00000000**	**-0.89528750**	**0.00000000**	**-0.85511480**
QueueAr	60	**0.00000000**	**-0.91825910**	**0.00000000**	**-0.93010370**
	120	**0.00000000**	**-0.88655810**	**0.00000000**	**-0.90334330**
	300	**0.00000000**	**-0.89079050**	**0.00000000**	**-0.90928440**
StringTokenizer	60	**0.00000000**	**-0.83481400**	**0.00000000**	**-0.84382940**
	120	**0.00000001**	**-0.76904390**	**0.00000000**	**-0.78520930**
	300	**0.00000000**	**-0.79787330**	**0.00000000**	**-0.80784360**

Amenazas a la validez interna

Las amenazas a la validez interna surgen a partir de la forma en que realizamos los experimentos y refieren al impacto de esto en los resultados obtenidos.

Amenaza 2. *Las diferencias inducidas en los sujetos mediante mutaciones, para el caso de los experimentos de detección de transiciones y sujetos inválidos, están sesgadas, y favorecen a nuestro prototipo como herramienta de detección*

Para mitigar esta amenaza, se generaron todos los mutantes posibles con la herramienta muJava (herramienta estándar independiente de EvoSuite), y se eligieron aleatoriamente 50 de estos mutantes.

Amenaza 3. *Las EPAs de los sujetos experimentales utilizadas no modelan correctamente el comportamiento de los mismos*

Para atacar esta amenaza, se tuvo especial cuidado en la creación de las EPAs. El hecho de que se trate de componentes relativamente simples nos brinda más confianza acerca de la correctitud de las EPAs creadas. Además, se utilizó nuestro prototipo para intentar detectar transiciones inválidas en los sujetos originales, y no pudo encontrarse ninguna.

Amenaza 4. *Las herramientas que se utilizaron para medir las coberturas de los conjuntos de casos de test contienen fallas o sesgos*

Para atacar esta amenaza intentamos realizar análisis de cobertura con otras herramientas. Para el caso de cobertura de líneas y ramas, intentamos utilizar JCov y Cobertura. Ambas herramientas conflictuaban con los tests generados por EvoSuite, principalmente debido al *runtime* que precisan los casos de tests de EvoSuite para poder ejecutarse. En cuanto a mutation testing, comprobamos que los mutation scores observados eran similares a los dados por la herramienta PIT.

Tab. 4.19: Correlaciones según el coeficiente de Pearson entre cobertura de transiciones de EPA y cobertura de código (líneas y ramas) para los conjuntos de casos de test generados con EvoSuite con `LINE` y `BRANCH` (EvoSuite `L+B`), y EvoSuite con `LINE`, `BRANCH`, `EPAERROR` y `EPATRANSITION` (EvoSuite `L+B+EPA+ERR`).

Subject	Budget	p-valor	Corr EPA,LINE	p-valor	Corr EPA,BRNCH
ListIterator	60	nan	nan	nan	nan
	120	nan	nan	nan	nan
	300	0.29652000	-0.16923110	0.29652000	-0.16923110
BoundedStack	60	nan	nan	nan	nan
	120	nan	nan	nan	nan
	300	nan	nan	nan	nan
NFST	60	0.31762450	-0.16210340	0.31762450	-0.16210340
	120	0.31075540	0.16438940	0.31075540	0.16438940
	300	nan	nan	nan	nan
QueueAr	60	nan	nan	nan	nan
	120	nan	nan	nan	nan
	300	nan	nan	nan	nan
StringTokenizer	60	nan	nan	0.30195970	-0.16736380
	120	nan	nan	nan	nan
	300	nan	nan	nan	nan

5. CONCLUSIONES Y TRABAJO FUTURO

En este trabajo, agregamos a EvoSuite el soporte necesario para poder emplear dos nuevos criterios a la hora de generar conjuntos de casos de test: `EPATRANSITION`, el criterio mediante el que se busca maximizar la cobertura de la EPA asociada al sujeto, y `EPAERROR`, el criterio que busca maximizar la cantidad de transiciones inválidas (no contempladas en la EPA) del mismo.

A partir de esta herramienta, realizamos experimentos con la intención de poner a prueba nuestro prototipo, corroborar resultados obtenidos en otros trabajos y confirmar o descartar nuestras propias hipótesis.

Encontramos que nuestro prototipo se mostró más efectivo que la herramienta original a la hora de generar conjuntos de casos de test con una alta cobertura de EPA en nuestros sujetos de prueba; en algunos casos, esta diferencia fue de casi un 70 %.

Al intentar corroborar las hipótesis, provenientes de investigación previa [1], de que mayores coberturas de EPA se condecirían con mayores coberturas de líneas y ramas, así como una mejora en la detección de fallas, encontramos que, al menos para el caso de la detección de fallas y para los casos que formaron parte de este trabajo, esto depende fuertemente de la herramienta utilizada para generar los conjuntos de casos de test.

Por otro lado, mostramos que nuestro prototipo pudo ser utilizado de manera efectiva para detectar diferencias entre el código y su EPA asociada para los sujetos involucrados en este estudio. La herramienta no sólo permite detectar que hay diferencias, sino que además construye un caso de test en el que se describe cómo ejecutar la transición inválida a la vez que muestra de qué transición se trata.

Trabajos futuros podrían ampliar el conjunto de sujetos. Resultaría de interés observar qué resultados se obtienen con sujetos más complejos, con EPAs más grandes en cantidad de estados y transiciones. En particular, si en sujetos más complejos se sostiene la efectividad del prototipo en cuanto a la obtención de altas coberturas de EPA.

Además, sería interesante conocer los resultados del uso de la herramienta que construimos para detectar transiciones de EPA inválidas y, por ende la presencia de diferencias entre código y protocolo, en casos reales, dado que nuestra experimentación se ciñó a sujetos generados mediante mutaciones realizadas con muJava.

Bibliografía

[1] H. Czemerinski, V. Braberman, and S. Uchitel, "Behaviour abstraction coverage as black-box adequacy criteria," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, March 2013, pp. 222–231.

[2] B. Hetzel, *The Complete Guide to Software Testing*, 2nd ed. Wellesley, MA, USA: QED Information Sciences, Inc., 1988.

[3] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley, 11 2011. [Online]. Available: http://amazon.com/o/ASIN/1118031962/

[4] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000-2014)," in *FOSE 2014, Hyderabad, India, May*, pp. 117–132.

[5] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 504–527. [Online]. Available: http://dx.doi.org/10.1007/11531142_22

[6] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[7] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*. ECSEL, Oct. 1999, pp. 21–28.

[8] G. Fraser and A. Arcuri, "Evosuite at the sbst 2016 tool competition," in *9th International Workshop on Search-Based Software Testing (SBST'16) at ICSE'16*, 2016, pp. 33–36.

[9] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[10] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel, "Program abstractions for behaviour validation," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 381–390. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985846

[11] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, Nov 2010.

[12] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Inf. Softw. Technol.*, vol. 51, no. 6, pp. 957–976, Jun. 2009. [Online]. Available: http://0-dx.doi.org.millenium.itesm.mx/10.1016/j.infsof.2008.12.005

[13] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.14

[14] F. Lammermann, A. Baresel, and J. Wegener, "Evaluating evolutionary testability for structure-oriented testing with software measurements," *Appl. Soft Comput.*, vol. 8, no. 2, pp. 1018–1028, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.asoc.2006.06.010

[15] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, pp. 1–42, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9424-2

[16] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*. Springer, 2015, pp. 93–108.

[17] "Contractor validation tool." [Online]. Available: http://lafhis.dc.uba.ar/dependex/contractor/Welcome.html

[18] "Tutorial part 4: Extending evosuite." [Online]. Available: http://www.evosuite.org/documentation/tutorial-part-4/

[19] M. A. Weiss, *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1998.

[20] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 178–189. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635890

[21] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, May 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1486

[22] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[23] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831728

[24] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Softw. Engg.*, vol. 20, no. 3, pp. 783–812, Jun. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9299-z

[25] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Routledge, 1988.