

# Generación automatizada de pruebas unitarias para Python

Stephan Lukasczyk, Florian Kroiß y Gordon Fraser

Universidad de Passau, Innstr. 33, 94032 Passau, Alemania  
{stephan.lukasczyk,gordon.fraser}@uni-passau.de, kroiss@fim.uni-passau.de

**Resumen** La generación automatizada de pruebas unitarias es un campo de investigación establecido, y existen herramientas maduras de generación de pruebas para lenguajes de programación tipificados estáticamente como Java. Sin embargo, es sustancialmente más difícil generar automáticamente pruebas de apoyo para lenguajes de programación tipificados dinámicamente como Python, debido a la falta de información de tipo y la naturaleza dinámica del lenguaje. En este artículo, describimos una incursión en el problema de la generación de pruebas unitarias para lenguajes tipificados dinámicamente. Presentamos Pynguin, un marco de generación de pruebas unitarias automatizadas para Python. Usando Pynguin, nuestro objetivo es arrojar luz empíricamente sobre dos preguntas centrales: (1) ¿Los métodos de generación de pruebas basados en búsquedas bien establecidos, previamente evaluados solo en lenguajes escritos estáticamente, se generalizan a lenguajes escritos dinámicamente? (2) ¿Cuál es la influencia de la información de tipos incompletos y la tipificación dinámica en el problema de la generación de pruebas automatizadas? Nuestros experimentos confirman que los algoritmos evolutivos pueden superar la generación de pruebas aleatorias también en el contexto de Python, e incluso pueden aliviar el problema de la ausencia de información de tipo hasta cierto punto. Sin embargo, nuestros resultados demuestran que la tipificación dinámica plantea un problema fundamental para la generación de pruebas, lo que sugiere un trabajo futuro sobre la integración de la inferencia de tipos.

**Palabras clave:** Escritura dinámica · Python · Generación aleatoria de pruebas · Generación de prueba de toda la suite

## 1. Introducción

Las pruebas unitarias se pueden generar automáticamente para ayudar a los desarrolladores y al análisis dinámico de los programas. Las técnicas establecidas, como la generación de pruebas aleatorias dirigidas por retroalimentación [15] o los algoritmos evolutivos [7], se implementan en prototipos de investigación maduros, pero se basan en suposiciones sólidas sobre la disponibilidad de información de tipo estático, como es el caso en lenguajes tipificados estáticamente como Java. Sin embargo, los lenguajes de escritura dinámica, como Python o JavaScript, han aumentado su popularidad en los últimos años. Python es el lenguaje de programación más popular en la categoría de lenguajes de escritura a, por ejemplo, el aprendizaje automático y el <sup>1</sup> dinámica, según . Es muy utilizado en los campos de análisis de datos de IEEE Spectrum Ranking , y también es popular en otros dominios. Este

<sup>1</sup> <https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019> , consultado el 25 de julio de 2020.

se puede ver, por ejemplo, en el Python Package Index (PyPI), que contiene más de 200 000 paquetes en el momento de escribir este artículo. En lenguajes como Python, la información de tipo que requieren los generadores de pruebas unitarias automatizadas no está disponible.

Un generador de prueba de unidad automatizado requiere principalmente información de tipo para seleccionar parámetros para llamadas de función y generar objetos complejos. Si la información de tipo está ausente, el generador de pruebas solo puede adivinar qué llamadas usar para crear nuevos objetos o qué objetos existentes seleccionar como parámetros para nuevas llamadas a funciones. Por lo tanto, los generadores de pruebas existentes para lenguajes tipeados dinámicamente recurren a otros medios para evitar tener que tomar tales decisiones en primer lugar, por ejemplo, utilizando el modelo de objeto de documento de un navegador web para generar pruebas para JavaScript [14], o apuntando al navegador. sistema de manejo de eventos en lugar de API [3, 12]. Sin embargo, todavía no existe un generador de pruebas unitarias de propósito general a nivel de API para lenguajes como Python.

Con el fin de permitir que la investigación de generación de pruebas amplíe su enfoque de lenguajes tipificados de forma estática a dinámica, en este documento presentamos Pynguin, un nuevo marco de generación de pruebas automatizadas para Python. Pynguin toma como entrada un módulo de Python y sus dependencias, y tiene como objetivo generar automáticamente pruebas unitarias que maximicen la cobertura del código. Para lograr esto, Pynguin implementa las técnicas establecidas de generación de pruebas de generación de conjuntos completos [9] y generación aleatoria dirigida por retroalimentación [15]. Pynguin está disponible como código abierto para respaldar futuras investigaciones sobre la generación automatizada de pruebas para lenguajes de programación tipificados dinámicamente. Pynguin está diseñado para ser extensible; en este documento nos enfocamos en algoritmos de referencia establecidos para experimentos fundamentales, agregaremos más algoritmos como DynaMOSA [16] en trabajos futuros.

Usando Pynguin, estudiamos empíricamente el problema de la generación automatizada de pruebas unitarias para Python usando diez proyectos populares de código abierto de Python tomados de GitHub, todos los cuales contienen información de tipo agregada por los desarrolladores en términos de anotaciones de tipo. Esta selección nos permite estudiar dos preguntas centrales:

- (1) ¿Los hallazgos anteriores, que muestran que la búsqueda evolutiva logra una mayor cobertura de código que las pruebas aleatorias [5], también se generalizan a los lenguajes tipificados dinámicamente?
- (2) ¿Cuál es la influencia de la falta de información de tipos en un lenguaje tipado dinámicamente como Python en la generación de pruebas unitarias automatizadas?

En detalle, las contribuciones de este trabajo son las siguientes:

1. Presentamos Pynguin, un nuevo marco para la generación automatizada de pruebas unitarias para el lenguaje de programación Python.
2. Reproducimos experimentos realizados previamente solo en el contexto de lenguajes tipificados estáticamente para comparar enfoques de generación de pruebas.
3. Estudiamos empíricamente la influencia del tipo de información en la efectividad de generación de pruebas automatizadas.

Nuestros experimentos confirman que el enfoque de conjunto completo generalmente logra una mayor cobertura de código que las pruebas aleatorias, y que la disponibilidad de información de tipo también conduce a una mayor cobertura resultante. Sin embargo, nuestros experimentos revelan varios desafíos técnicos nuevos, como generar colecciones o tipos de entrada iterables. Nuestros hallazgos también sugieren que la integración de la investigación actual sobre la inferencia de tipos es una ruta prometedora para futuras investigaciones.

## 2. Fondo

Los principales enfoques para generar automáticamente pruebas unitarias son mediante la creación de secuencias aleatorias o mediante la aplicación de algoritmos de búsqueda metaheurísticos. Las pruebas aleatorias ensamblan secuencias de llamadas a constructores y métodos aleatoriamente, a menudo con el objetivo de encontrar excepciones no declaradas [6] o violaciones de contratos generales de objetos [15], pero las pruebas generadas también se pueden usar como pruebas de regresión automatizadas. La eficacia de los generadores de pruebas aleatorias se puede aumentar mediante la integración de heurísticas [13, 17]. Los enfoques basados en la búsqueda utilizan una representación similar, pero aplican algoritmos de búsqueda evolutivos para maximizar la cobertura del código [1, 4, 9, 19].

Como ejemplo para ilustrar cómo los generadores de prueba existentes utilizan la información de tipos, considere los siguientes fragmentos de código Java (izquierda) y Python (derecha):

```
class Foo { Foo
  ( Bar b ) { ... } void doFoo ( Bar
    b ) { ... } } class Bar { Bar () { ... }

  Bar doBar ( Bar b ) { ... } }
```

```
class Foo:
  def __init__(auto def do_foo , b ) : ...
    (auto , b ) : ...
  barra de clase :
  def __init__(auto): ... def do_bar (auto
    b): ... ,
```

Suponga que Foo del ejemplo de Java es la clase bajo prueba. Tiene una dependencia de la clase Bar: para generar un objeto de tipo Foo necesitamos una instancia de Bar, y el método doFoo también requiere un parámetro de tipo Bar.

La generación de pruebas aleatorias normalmente generaría pruebas de forma directa. Comenzando con una secuencia vacía  $t_0 = \text{hola}$ , se pueden seleccionar todas las llamadas disponibles para las cuales todos los parámetros pueden cumplirse con objetos que ya existen en la secuencia. En nuestro ejemplo, inicialmente solo se puede llamar al constructor de Bar, ya que todos los demás métodos y constructores requieren un parámetro, lo que da como resultado  $t_1 = \text{ho1} = \text{new Bar}()$ .i. Dado que  $t_1$  contiene un objeto de tipo Bar, en el segundo paso, el generador de pruebas ahora tiene la opción de invocar doBar en ese objeto (y usar el mismo objeto también como parámetro) o invocar al constructor de Foo. Asumiendo que la llamada elegida es el constructor de Foo, ahora tenemos  $t_2 = \text{ho1} = \text{new Bar}()$ ;  $\text{o2} = \text{nuevo Foo(o1)}$ ;i. Dado que ahora también hay una instancia de Foo en la secuencia, en el siguiente paso también el método doFoo es una opción. El generador de pruebas aleatorias continuará extendiendo la secuencia de esta manera, posiblemente integrando heurísticas para seleccionar llamadas más relevantes, o para decidir cuándo comenzar con una nueva secuencia, etc.

Un enfoque alternativo, por ejemplo aplicado durante el paso de mutación de un generador de prueba evolutivo, es seleccionar las llamadas necesarias de forma inversa.

Es decir, un generador de pruebas basado en búsqueda como EvoSuite [9] primero decidiría que necesita, por ejemplo, llamar al método doFoo de la clase Foo. Para lograr esto, requiere una instancia de Foo y una instancia de Bar para satisfacer las dependencias.

Para generar un objeto de parámetro de tipo Bar, el generador de pruebas consideraría todas las llamadas que se declaran para devolver una instancia de Bar, que es el caso del constructor de Bar en nuestro ejemplo, por lo que antepondría una llamada a Bar() antes la invocación de doFoo. Además, intentaría crear una instancia de Foo llamando al constructor. Esto, a su vez, requiere una instancia de Bar, para la cual el generador de pruebas podría usar la instancia existente o podría invocar al constructor de Bar.

En ambos escenarios, la información de tipo es crucial: en el tipo de construcción hacia adelante, la información se usa para informar la elección de la llamada para agregar a la secuencia, mientras que en el tipo de construcción hacia atrás, la información se usa para seleccionar generadores de objetos de dependencia. Sin información de tipo, como es el caso con el ejemplo de Python, una construcción directa (1) debe permitir todas las funciones posibles en todos los pasos, por lo que no solo puede seleccionar el constructor de Bar, sino también el de Foo con un tipo de parámetro arbitrario, y (2) tiene que considerar todos los objetos existentes para todos los parámetros de una llamada seleccionada, y así, por ejemplo, también str o int. La construcción hacia atrás sin información de tipo también tendría que intentar seleccionar generadores de todas las llamadas posibles y todos los objetos posibles, lo que da como resultado un espacio de búsqueda potencialmente grande para seleccionar.

La información de tipo se puede proporcionar de dos maneras en las versiones recientes de Python: ya sea en un archivo de código auxiliar que contiene sugerencias de tipo o directamente anotado en el código fuente. Un archivo de resguardo se puede comparar con los archivos de encabezado de C: contienen, por ejemplo, declaraciones de métodos con sus tipos correspondientes. Desde Python 3.5, los tipos también se pueden anotar directamente en el código fuente de implementación, de manera similar a los lenguajes tipificados estáticamente (ver PEP 484).

### 3 Generación de pruebas unitarias basadas en búsquedas

#### 3.1 Generación de pruebas de Python como un problema de búsqueda

Como unidad para la generación de pruebas unitarias, consideramos los módulos de Python. Un módulo suele ser idéntico a un archivo y contiene definiciones de, por ejemplo, funciones, clases o sentencias; estos se pueden anidar casi arbitrariamente. Cuando se carga el módulo, se ejecutan las definiciones y declaraciones en el nivel superior. Al generar pruebas, no solo queremos que se ejecuten todas las definiciones, sino también todas las estructuras definidas por esas definiciones, por ejemplo, funciones, cierres o listas de comprensión. Por lo tanto, para aplicar un algoritmo de búsqueda, primero debemos definir una representación adecuada de las soluciones válidas para este problema.

Usamos una representación basada en trabajos previos del dominio de prueba de código Java [9]. Para cada enunciado  $s_j$  en un caso de prueba  $t_i$  asignamos un valor  $v(s_j)$  con tipo  $\tilde{v}(s_j)$  y  $T$  con el conjunto finito de tipos, tal que  $\tilde{v}(s_j) \in T$ . Los tipos  $\tilde{v}(s_j)$  se definen por primitivas representan variables int, float, bool y str, por ejemplo,  $\text{var0} = 42$ . El valor y el tipo de una declaración están definidos por la variable primitiva. Tenga en cuenta que aunque en Python todo es un objeto, tratamos estos valores como primitivos porque no requieren una mayor construcción en la sintaxis de Python. Otros tipos simples, como las listas, requieren la construcción de la lista y sus elementos, que aún no manejamos. Las instrucciones del constructor crean nuevas instancias de una clase, por ejemplo,  $\text{var0} = \text{SomeType}()$ . El valor y el tipo están definidos por el objeto construido; cualquier parámetro se satisface del conjunto  $V = \{v(s_k) \mid 0 \leq k < j\}$ . Las sentencias de métodos invocan métodos en objetos, por ejemplo,  $\text{var1} = \text{var0.foo}()$ . El valor y el tipo están definidos por el valor de retorno.

<sup>2</sup> <https://python.org/dev/peps/pep-0484/>, consultado el 25 de julio de 2020.

del método; el objeto fuente y todos los parámetros se satisfacen a partir del conjunto V. Las sentencias de función invocan funciones, por ejemplo, `var2 = bar()`. No requieren un objeto de origen, pero por lo demás son idénticos a las instrucciones de método. Esta representación es de tamaño variable; restringimos el tamaño de los casos de prueba  $l$  y  $[1, L]$  y los conjuntos de prueba  $n$  y  $[1, N]$ . A diferencia del trabajo anterior sobre la prueba de Java [9], no definimos declaraciones de campo o asignación; Los campos de los objetos no se declaran explícitamente en Python, sino que se asignan dinámicamente, por lo que no es trivial identificar los campos existentes de un objeto y lo dejamos como trabajo futuro.

Los operadores de búsqueda para esta representación se basan en los utilizados en EvoSuite [9]: Crossover toma como entrada dos suites de prueba  $P1$  y  $P2$ , y genera dos descendientes  $O1$  y  $O2$ . Los casos de prueba individuales no tienen dependencias entre sí, por lo que la aplicación del cruce siempre genera suites de prueba válidas como descendencia. Además, el operador reduce la diferencia en el número de casos de prueba entre los conjuntos de pruebas, por lo que  $\text{abs}(|O1| - |O2|)$  y  $\text{abs}(|P1| - |P2|)$ .

Por lo tanto, ningún descendiente tendrá más casos de prueba que el más grande de sus padres.

Al mutar un conjunto de pruebas  $T$ , cada uno de sus casos de prueba se muta con probabilidad  $\frac{1}{|T|}$ . Después de la mutación, se agregan nuevos casos de prueba. Si se agregan  $n$  casos de prueba, se agregan  $n$  casos de prueba nuevos con probabilidad  $\frac{1}{n}$  y se ha alcanzado, por lo tanto  $|T|$  y  $N$ . La mutación de un caso de prueba puede ser en  $\frac{1}{|T|}$  de los casos de prueba, que elimina una declaración del caso de prueba, que agrega o elimina caracteres, que agrega o elimina caracteres o cambiando llamadas a métodos) e insertar, que agrega nuevas declaraciones en posiciones aleatorias en el caso de prueba. Cada una de estas operaciones puede suceder con la misma probabilidad de  $\frac{1}{3}$ . Un caso de prueba que no tiene declaraciones después de la aplicación del operador de mutación se elimina del conjunto de pruebas  $T$ . Para construir la población inicial, un caso de prueba aleatorio  $t$  se muestra  $\frac{1}{|T|}$  veces uniformemente, luego aplicando el operador de inserción repetidamente comenzando con un caso de prueba vacío  $t$  y  $r$ .

$^{\circ}$ , hasta  $|t|^{\circ}$

### 3.2 Cobertura del código Python

Un módulo de Python contiene varias estructuras de control, por ejemplo, declaraciones `if` o `while`, que están protegidas por predicados lógicos. Las estructuras de control están representadas por saltos condicionales a nivel de bytecode, basados en un predicado unario o binario. Nos enfocamos en la cobertura de ramas en este trabajo, lo que requiere que cada uno de esos predicados se evalúe como verdadero y falso.

Sea  $B$  el conjunto de ramas en el SUT: dos por cada salto condicional en el código de bytes. Todo lo ejecutable en Python se representa como un objeto de código. Por ejemplo, un módulo completo se representa como un objeto de código, una función dentro de ese módulo se representa como otro objeto de código. Queremos ejecutar todos los objetos de código  $C$  del SUT. Por lo tanto, hacemos un seguimiento de los objetos de código  $CT$  ejecutados, así como de la distancia mínima de bifurcación  $d_{\min}(b, T)$  para cada bifurcación  $b$  y  $B$ , cuando se ejecuta un conjunto de pruebas  $T$ .  $BT$  y  $B$  denota el conjunto de bifurcaciones tomadas. Luego definimos la cobertura de rama  $\text{cov}(T)$  de un conjunto de pruebas  $T$  como  $\text{cov}(T) = \frac{|CT| + |BT|}{|C| + |B|}$ .

La función de aptitud requerida por el algoritmo genético de nuestro enfoque de conjunto completo se construye de manera similar a la utilizada en EvoSuite [9] al incorporar la distancia de rama. La distancia de rama es una heurística para determinar qué tan lejos está un predicado de evaluarse como verdadero o falso, respectivamente. A diferencia del trabajo anterior en Java, donde la mayoría de los predicados en el nivel de bytecode operan solo en valores booleanos o numéricos, en nuestro caso los operandos de un predicado pueden ser cualquier objeto de Python. Por lo tanto, como señaló Arcuri [2], tenemos que definir nuestra distancia de rama de tal manera que pueda manejar objetos de Python arbitrarios.

Sea  $O$  el conjunto de posibles objetos de Python y sea  $\tilde{y} := \{\tilde{y}, 6\tilde{y}, <, \tilde{y}, >, \tilde{y}, \tilde{y}, / \tilde{y}, =, 6=\}$  el conjunto de operadores de comparación binaria (observación: usamos ' $\tilde{y}$ ', ' $=$ ' y ' $\tilde{y}$ ' para las palabras clave `==`, `is` y `in` de Python, respectivamente). Para cada  $\tilde{y} \in \tilde{y}$ , definimos una función  $\tilde{y} : O \times O \rightarrow \mathbb{R}_{\geq 0}$  que calcula la distancia de rama de  $\tilde{y}$  con  $a, b \in O$  y  $\tilde{y} \in \tilde{y}$ . Por  $\tilde{y}^-(a, b)$  denotamos la distancia de la rama falsa, donde  $\tilde{y}$  es el operador complementario de  $\tilde{y}$ . Sea además  $k$  un número positivo, y sea  $\text{lev}(x, y)$  la distancia de Levenshtein [11] entre dos cadenas  $x$  e  $y$ . Los predicados `is_numeric(z)` y `is_string(z)` determinan si el tipo de su argumento  $z$  es numérico o una cadena, respectivamente.

$$\begin{aligned} \tilde{y} \tilde{y}(a, b) &= \begin{array}{ll} \tilde{y} \ 0 & \text{un } \tilde{y} \text{ segundo} \\ |a \tilde{y} b| \ a \ 6\tilde{y} \ b \ \tilde{y} \ \text{is\_numeric}(a) \ \tilde{y} \ \text{is\_numeric}(b) \ \text{lev}(a, \\ \text{---} & \text{---} \\ b) \ a \ 6\tilde{y} \ b \ \tilde{y} \ \text{is\_string}(a) \ \tilde{y} \ \text{is\_string}(b) \\ \text{---} & \text{---} \\ \tilde{y} \text{ de lo contrario} & \end{array} \\ \tilde{y} <(a, b) &= \begin{array}{ll} \tilde{y} \ 0 & \text{un } < \text{ segundo} \\ a \ \tilde{y} \ b + k a \ \tilde{y} \ b \ \tilde{y} \ \text{es\_numérico}(a) \ \tilde{y} \ \text{es\_numérico}(b) \\ \text{---} & \text{---} \\ \tilde{y} & \text{de lo contrario} \end{array} \\ \tilde{y} \tilde{y} \tilde{y}(a, b) &= \begin{array}{ll} \tilde{y} \tilde{y} \ \tilde{y} \ 0 & a \ \tilde{y} \\ \text{---} & \text{---} \\ b a \ \tilde{y} \ b + k a \ > \ b \ \tilde{y} \ \text{es\_numérico}(a) \ \tilde{y} \ \text{es\_numérico}(b) \\ \text{---} & \text{---} \\ \tilde{y} & \text{de lo contrario} \end{array} \\ \tilde{y} \tilde{y} \tilde{y} >(a, b) &= \tilde{y} <(b, \\ a) \ \tilde{y} \tilde{y}(a, b) &= \tilde{y} \tilde{y}(b, a) \\ \tilde{y} \tilde{y}(a, \text{segundo}) &= \begin{array}{ll} k \text{ de lo contrario} & \tilde{y} \ \tilde{y} \ \{6\tilde{y}, \tilde{y}, / \tilde{y}, =, 6=\} \\ 0 \text{ un } \tilde{y} \text{ segundo} & \end{array} \end{aligned}$$

Tenga en cuenta que cada objeto en Python representa un valor booleano y, por lo tanto, puede usarse como predicado. Asignamos una distancia de 0 a la rama verdadera de tal predicado unario, si el objeto representa un valor verdadero, de lo contrario  $k$ . El trabajo futuro deberá refinar la distancia de rama para diferentes operadores y tipos de operandos.

La función de aptitud estima qué tan cerca está un conjunto de pruebas de cubrir todas las ramas del SUT. Por lo tanto, cada predicado debe ejecutarse al menos dos veces, lo cual

cumplir de la misma manera que el trabajo existente [9]: la distancia real de la rama  $d(b, T)$  viene dada por

$$d(b, T) = \begin{cases} 0 & \text{si la rama ha sido cubierta} \\ \frac{1}{n} & \text{si el predicado se ha ejecutado al menos dos veces, de lo contrario} \end{cases}$$

con  $v(x) = \frac{x}{x+1}$  siendo una función de normalización [9].

Finalmente, podemos definir la función de aptitud resultante  $f$  de un conjunto de pruebas  $T$  como

$$f(T) = |C| \cdot \frac{1}{|T|} + \sum_{b \in B} v(d(b, T))$$

### 3.3 El marco Pynguin

Pynguin es un marco para la generación automatizada de pruebas unitarias escrito en y para el lenguaje de programación Python. El marco está disponible como software de código abierto con licencia bajo la Licencia Pública General Menor de GNU desde su GitHub repositorio<sup>3</sup>. También se puede instalar desde Python Package Index (PyPI)<sup>4</sup> usando la utilidad `pip`.

Pynguin toma como entrada un módulo de Python y permite generar pruebas unitarias utilizando diferentes técnicas. Para ello, analiza el módulo y extrae información sobre los métodos disponibles en el módulo y los tipos del módulo y sus importaciones. Hasta ahora, Pynguin se enfoca en la generación de entrada de prueba y excluye la generación de oráculos. La ejecución de una herramienta emite los casos de prueba generados al estilo del marco `PyTest`<sup>5</sup> ampliamente utilizado o para el módulo `unittest` de la biblioteca estándar de Python.

Pynguin está diseñado para ser extensible con otros enfoques y algoritmos de generación de pruebas. Para los experimentos de este documento, implementamos un enfoque aleatorio dirigido por retroalimentación basado en Randoop [15] además del enfoque de generación de pruebas de conjunto completo. La generación de pruebas dirigida por retroalimentación comienza con dos conjuntos de pruebas vacíos, un conjunto de pruebas aprobado y uno fallido, y agrega declaraciones aleatoriamente a un caso de prueba vacío. Después de cada adición, se ejecuta el caso de prueba y se recupera el resultado de la ejecución. Los casos de prueba exitosos, es decir, los casos de prueba que no generan excepciones se agregan al conjunto de pruebas aprobado; un caso de prueba que genera una excepción se agrega al conjunto de pruebas que falla. A continuación, el algoritmo elige aleatoriamente un caso de prueba del conjunto de pruebas aprobado o un caso de prueba vacío y le agrega declaraciones. Remitimos al lector a la descripción de Randoop [15] para obtener detalles sobre el algoritmo; Las principales diferencias de nuestro enfoque son que aún no verifica las violaciones del contrato y no requiere que el usuario proporcione una lista de clases y métodos relevantes, lo que hace Randoop.

<sup>3</sup> <https://github.com/se2p/pynguin>, consultado el 27 de julio de 2020. <https://>

<sup>4</sup> [pypi.org/project/pynguin/](https://pypi.org/project/pynguin/), consultado el 25 de julio de 2020. <https://www.pytest.org>,

<sup>5</sup> consultado el 25 de julio de 2020.

## 4 Evaluación Experimental

Usando nuestro generador de pruebas Pynguin, nuestro objetivo es estudiar empíricamente la generación automatizada de pruebas unitarias en Python. En primer lugar, nos interesa determinar si los hallazgos previos sobre el rendimiento de las técnicas de generación de pruebas establecidas en el contexto de los lenguajes tipificados estáticamente se generalizan también a Python:

Pregunta de investigación 1 (RQ1) ¿Cómo se comparan la generación de pruebas de conjunto completo y la generación de pruebas aleatorias en el código de Python?

Una diferencia central entre el trabajo anterior y el contexto de Python es la información de tipo: el trabajo anterior evaluó técnicas de generación de pruebas principalmente para lenguajes tipificados estáticamente, como Java, donde la información sobre tipos de parámetros está disponible en tiempo de compilación, es decir, sin ejecutar el programa. Este no es el caso de muchos programas escritos en lenguajes de escritura dinámica, como Python. Por lo tanto, queremos evaluar explícitamente la influencia de la información de tipo para el proceso de generación de pruebas:

Pregunta de investigación 2 (RQ2) ¿Cómo influye la disponibilidad de información de tipos en la generación de pruebas?

### 4.1 Configuración experimental

Para responder a las dos preguntas de investigación, creamos un conjunto de datos de proyectos de Python para la experimentación. Utilizamos la categoría 'escrito' del índice de paquetes PyPI de los proyectos de Python y seleccionamos diez proyectos mediante la búsqueda de proyectos que contienen sugerencias de tipo en sus firmas de método y que no tienen dependencias con bibliotecas de código nativo, como numpy. Los detalles de los proyectos elegidos se muestran en la Tabla 1: la columna Nombre del proyecto da el nombre del proyecto en PyPI; las líneas de código se midieron con la herramienta de utilidad cloc<sup>6</sup>. Además, la tabla muestra el número promedio absoluto de objetos de código, predicados y tipos detectados por módulo de cada proyecto. Las dos primeras medidas dan una idea de la complejidad del proyecto; los números más altos indican una mayor complejidad. Este último proporciona una descripción general de cuántos tipos Pynguin pudo analizar (tenga en cuenta que es posible que Pynguin no pueda resolver todos los tipos).

La métrica central que utilizamos para evaluar el rendimiento de una técnica de generación de pruebas es la cobertura de código. En particular, medimos la cobertura de sucursales a nivel de bytecode; Al igual que en el código de bytes de Java, las condiciones complejas se compilan en ramas anidadas con condiciones atómicas también en el código de Python. Además de la cobertura general final, también realizamos un seguimiento de la cobertura a lo largo del tiempo para arrojar luz sobre la velocidad de convergencia. Para comparar estadísticamente los resultados utilizamos la prueba U de Mann-Whitney y el tamaño del efecto de Vargha y Delaney  $A^*$ <sup>12</sup>

Ejecutamos Pynguin en cuatro configuraciones diferentes: Primero, ejecutamos Pynguin utilizando la generación de pruebas aleatorias y la generación de conjuntos de pruebas completos; segundo, ejecutamos Pynguin con las anotaciones de tipo escritas por el desarrollador contenidas en el

<sup>6</sup> <https://github.com/AIDanial/cloc>, consultado el 25 de julio de 2020.



Cuadro 1: Proyectos utilizados para la evaluación

Nombre del proyecto	Versión LOCs Módulos CodeObs. Pres. Tipos				
apimd	1.0.2 316 1.0.1	1	35,0	83,0	11,0
async_btree	284 1.2.0 85	6	9,0	8,7	6,3
codetiming	0.7.1 608	2	18,0	8,0	6,0
docstring_parser flautas	0.2.0.post0 085	6	12,0	15,7	9,5
	0.6 1 715	9	19,0	26,0	5,0
flutiles		13	10.2	22.3	8.4
mimetismo	4.0.0 1 663	34	12.3	5.7	9.2
pypara	0.0.22 1 305	6	47.2	23,5	12,0
python-string-utils pyutils	1.0.0 476	4	21.0	29.5	6.5
	0.4.1 1 108	23	8.2	6.6	6.1
Total	8 645	104	191,9	229,0	79,9

proyectos, y sin ellos. Para responder a RQ1, comparamos el rendimiento de generación de pruebas aleatorias y generación de conjunto completo de pruebas; para responder RQ2 nosotros comparar el rendimiento de cada una de estas técnicas para el caso con y sin información de tipo.

Para cada proyecto, Pynguin se ejecutó en cada uno de los módulos constituyentes en secuencia. Ejecutamos Pynguin en git revision 5f538833 en un contenedor Docker que se basa en Debian 10 y utiliza Python 3.8.3. En línea con lo anterior trabajo, establecemos el límite de tiempo máximo para los algoritmos de generación de prueba, que es decir, el tiempo sin analizar el módulo bajo prueba y sin exportar el resultados, a 600 s por módulo. Ejecutamos Pynguin 30 veces en cada módulo y configuración para minimizar la influencia de la aleatoriedad. Todos los experimentos fueron ejecutado en servidores informáticos dedicados equipados con Intel Xeon E5-2690v2 CPU y 64 GB de RAM, con Debian 10. Todos los scripts y los datos sin procesar son disponible como material complementario<sup>7</sup>.

#### 4.2 Amenazas a la Validez

**Validez interna** La herramienta de cobertura estándar para Python es Coverage.py, que ofrece la capacidad de medir la cobertura de sucursales. Sin embargo, mide rama cobertura comparando qué transiciones entre las líneas fuente se han producido y que son posibles. Este método para medir la cobertura de ramales es impreciso, porque no todas las declaraciones de ramales conducen necesariamente a una transición de línea fuente. por ejemplo,  $x = 0$  si  $y > 42$  sino 1337. Así implementamos nuestra propia medición de cobertura. Intentamos mitigar posibles errores en nuestra implementación, proporcionando suficientes pruebas unitarias para ello.

**Validez externa** Utilizamos 104 módulos de diez proyectos diferentes de Python para nuestros experimentos. Es concebible que la exclusión de proyectos sin tipo

<sup>7</sup> <https://github.com/se2p/artifact-pynguin-ssbse2020>, consultado el 27 de julio de 2020.

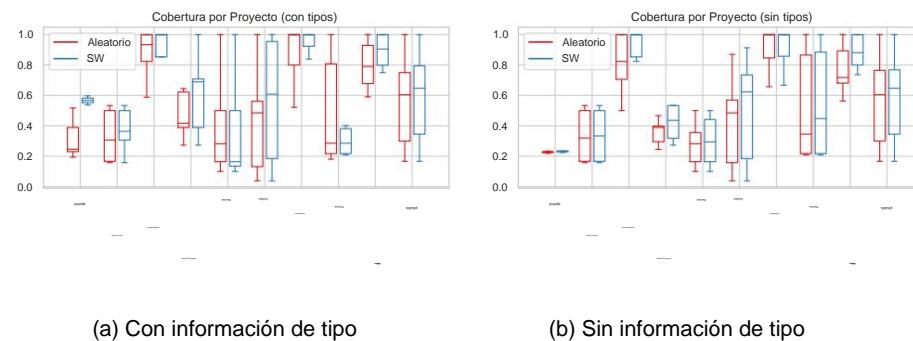


Figura 1: Cobertura por proyecto y configuración

Las anotaciones o las bibliotecas de código nativo conducen a una selección de proyectos más pequeños y, por lo tanto, es posible que los resultados no se generalicen a otros proyectos de Python. Sin embargo, además de las dos restricciones enumeradas, no se aplicaron otras durante la selección.

Los métodos de validez de construcción llamados con tipos de entrada incorrectos aún pueden cubrir partes del código antes de generar posibles excepciones debido a las entradas no válidas. De manera conservadora, incluimos toda la cobertura en nuestro análisis, lo que puede mejorar la cobertura para las configuraciones que ignoran la información de tipo y, por lo tanto, reducir el efecto que observamos. Sin embargo, no afecta nuestras conclusiones generales. Además, no podemos medir la capacidad de encontrar fallas ya que nuestra herramienta no genera aserciones, lo cual está explícitamente fuera del alcance de este trabajo.

#### 4.3 RQ1: Generación de pruebas de conjunto completo frente a pruebas aleatorias

La Figura 1 proporciona una descripción general de la cobertura lograda por proyecto en diagramas de caja. Cada punto de datos en el gráfico es un valor de cobertura lograda para uno de los módulos del proyecto. La figura 1a informa los valores de cobertura para la generación de pruebas aleatorias y de conjunto completo con sugerencias de tipo disponibles, mientras que la figura 1b informa lo mismo sin el uso de sugerencias de tipo para guiar la generación.

Los valores de cobertura varían de 0 % a 100 % según el proyecto. La cobertura lograda varía entre proyectos, con algunos proyectos que generalmente logran una alta cobertura (por ejemplo, python-string-utils, mimesis, codetiming) y otros presentan desafíos para Pynguin (por ejemplo, apmid, async\_btree, pypara, flutes). Por ejemplo, para el proyecto apimid sin información de tipo, la cobertura es ligeramente superior al 20 %, que es la cobertura que se logra con solo importar el módulo. En Python, cuando se importa un módulo, se ejecutan las declaraciones de importación del módulo, así como las definiciones de clase y método, y por lo tanto se cubren.

Tenga en cuenta que esto no ejecuta los cuerpos del método. Para otros proyectos con baja cobertura, Pynguin no puede generar entradas razonables, por ejemplo, funciones o colecciones de orden superior, debido a limitaciones técnicas.

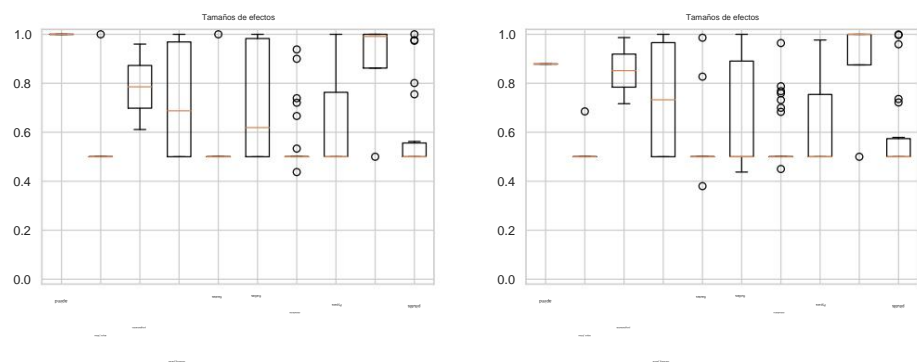
(a)  $A^{12}$  tamaños del efecto con información de tipo(b)  $A^{12}$  tamaños del efecto sin información de tipo

Figura 2: Tamaños del efecto de la suite completa frente a la generación aleatoria. Los valores superiores a 0,500 indican que todo el conjunto es mejor que aleatorio.

Para comprender mejor si la generación de pruebas de conjunto completo funciona mejor que la generación de pruebas aleatorias, la figura 2a informa los tamaños del efecto  $A^{12}$  para los dos con información de tipo disponible, mientras que la figura 2b informa lo mismo sin información de tipo disponible. En ambos diagramas de caja, un valor superior a 0,500 significa que el conjunto completo funciona mejor que la generación de pruebas aleatorias, es decir, produce resultados de mayor cobertura. Ambos gráficos muestran que, en promedio, toda la suite no funciona peor que el azar y, según el proyecto, puede lograr mejores resultados en términos de cobertura ( $A^{12}$  promedio con información de tipo: 0.618, sin información de tipo: 0.603). El efecto de estas mejoras es significativo ( $p < 0,05$ ) para seis de cada diez proyectos, sobre todo para apimd ( $A^{12} = 1,00$ , valor de  $p < 0,001$ ), python-string-utils ( $A^{12} = 0,705$ , valor de  $p < 0,001$ ) y el tiempo de código ( $A^{12} = 0,636$ , valor  $p = 0,005$ ).

Para los otros proyectos el efecto es insignificante. En el caso de la mimesis ( $A^{12} = 0,530$ ), esto se debe a los altos valores de cobertura en todas las configuraciones: la mayoría de los parámetros de método esperan tipos primitivos, que también se utilizan para la generación de entrada si no se proporciona información de tipo. Otros proyectos requieren habilidades técnicas específicas, por ejemplo, la mayoría de los métodos en `random` y `random2` (A^12 = 0,530) y `random3`, puede generar actualmente. La consecuencia de estas limitaciones técnicas es que Pynguin no puede alcanzar una mayor cobertura independientemente del algoritmo utilizado en estos casos. Observamos que los métodos bajo prueba a menudo requieren tipos de colección como entradas, en Python predominantemente listas y diccionarios. Además, generar estos tipos de entrada nos permitiría ejecutar más partes del código, lo que conduciría a una mayor cobertura y, por lo tanto, a mejores resultados. Dejamos esto, sin embargo, como trabajo futuro.

Otra limitación actual de nuestro marco radica en cómo se procesa la información de tipo disponible. Pynguin actualmente solo puede generar entradas para concreto

12 S. Lukaszcyk et al.

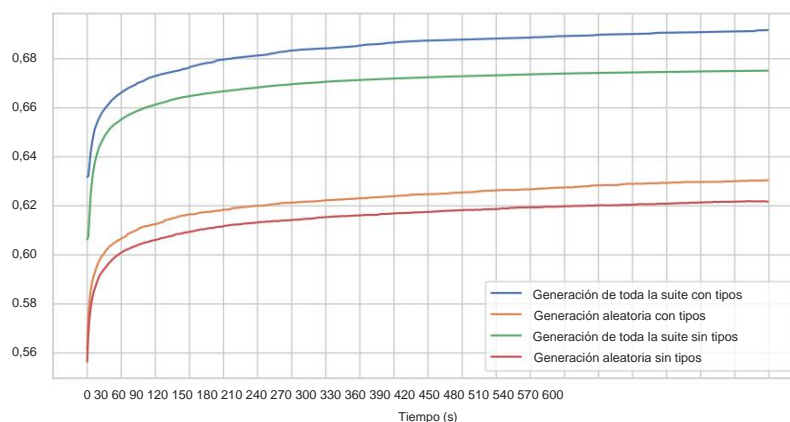


Figura 3: Evolución de la cobertura media a lo largo del tiempo

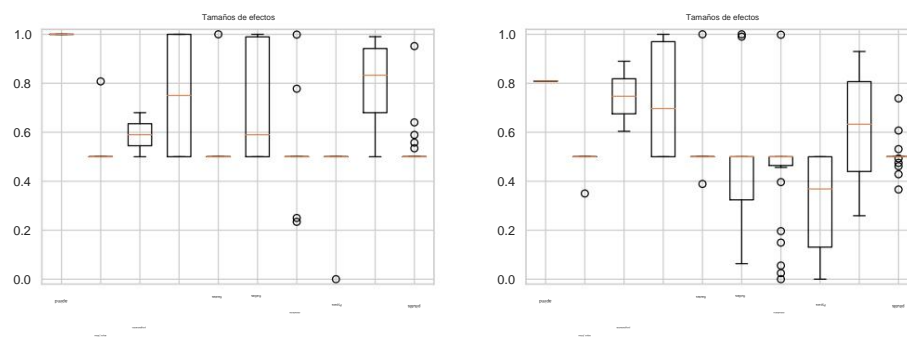
tipos, uniones de tipos y el tipo Cualquiera, para el cual intenta usar un tipo aleatorio del grupo de tipos disponibles en el SUT. El trabajo futuro se ocupará de las relaciones de subtipificación, así como de los tipos genéricos [10]. Otro tipo de parámetro frecuente que limita nuestra herramienta actual son las llamadas, es decir, funciones de orden superior que se pueden usar, por ejemplo, como devoluciones de llamadas. Trabajos previos han demostrado que generar funciones de orden superior como tipos de entrada es factible para lenguajes tipificados dinámicamente y beneficioso para la generación de pruebas [18]. Además, Pynguin actualmente solo tiene una estrategia de siembra estática ingenua para constantes que incorpora todos los valores constantes del proyecto bajo prueba en la generación de prueba, mientras que se ha demostrado que la siembra tiene una influencia positiva en la calidad de la generación de prueba [8] ya que permite una mejor -valores de entrada adecuados.

La figura 3 muestra la evolución de la cobertura media de todos los módulos durante el tiempo de generación disponible de 600 s. El gráfico de líneas indica claramente que la generación de conjuntos completos logra una mayor cobertura que la generación aleatoria, lo que nuevamente respalda nuestra afirmación. En general, podemos responder a nuestra primera pregunta de investigación de la siguiente manera:

Resumen (RQ1): la generación de pruebas de conjunto completo logra al menos una cobertura tan alta como la generación de pruebas aleatorias. Dependiendo del proyecto, logra una cobertura de moderada a fuertemente mayor.

#### 4.4 RQ2: Influencia de la Información Tipo

Para responder a RQ2, comparamos los valores de cobertura entre las configuraciones con y sin anotaciones de tipo, nuevamente utilizando los tamaños del efecto  $A^*$  para medir la cobertura. Esta vez, mostramos el efecto de la información de tipo para la generación de conjuntos completos en la Fig. 4a y para la generación aleatoria en la Fig. 4b. Para la generación de toda la suite, observamos un gran efecto positivo en algunos módulos, y casi ningún



(a)  $A^{12}$  tamaños del efecto para géneros de conjuntos completos

(b)  $A^{12}$  tamaños del efecto para la generación aleatoria

Figura 4: Tamaños de efecto para influencias de tipo

efecto para otros módulos cuando se incorpora información de tipo; reportamos un valor de 0.578 a favor promedio  $A^{12}$  del tipo de información. Para la generación aleatoria, notamos efectos similares, excepto para el proyecto pypara ( $A^{12} = 0.411$ ,  $p\text{-value} = 0.00937$ ); inspeccionar el código fuente de `pypara` de clase abstracta como anotaciones de tipo. Pynguin intenta crear una instancia de la clase abstracta, lo que falla y, por lo tanto, no puede generar entradas de método para gran parte del código porque no puede encontrar un subtipo instanciable. En general, sin embargo, reportamos un valor de 0.554 a favor de la generación aleatoria con información de tipo.

promedio  $A^{12}$

Los diagramas de caja en la Fig. 4 indican conclusiones similares para las pruebas aleatorias y de conjunto completo: la disponibilidad de información de tipo es beneficiosa para algunos proyectos, mientras que su efecto es insignificante para otros proyectos. El proyecto `docstring_parser`, por ejemplo, requiere sus propios tipos personalizados como valores de parámetro para muchos métodos. Sin información de tipo, Pynguin tiene que elegir aleatoriamente tipos de todos los tipos disponibles, con una baja probabilidad de elegir el correcto, mientras que con la información de tipo disponible puede generar directamente un objeto del tipo correcto. Otro efecto surge para los proyectos `python-string-utils`: la mayoría de sus métodos solo requieren tipos de entrada primitivos pero valores de entrada muy específicos. Pynguin utiliza una heurística de siembra constante estática simple para la generación de valores de entrada. Debido a que hay muchos valores en el grupo constante, la posibilidad de elegir el valor correcto es menor cuando no se conoce el tipo solicitado, lo que lleva a una cobertura más baja sin información de tipo.

Por otro lado, proyectos como `flutes` requieren iterables y callables como parámetros en muchos casos o necesitan un tratamiento especial de sus métodos para ejecutarlos correctamente (ver `coroutines` en `async_btrees`, por ejemplo). Pynguin actualmente carece de soporte para generar estos tipos requeridos, lo que evita efectos mayores pero no limita el enfoque general. Por lo tanto, la información de tipo no se puede usar de manera efectiva, lo que da como resultado efectos insignificantes entre las configuraciones comparadas.

El diagrama de líneas de la Fig. 3 muestra la cobertura promedio por configuración evaluada durante el tiempo disponible para la generación de pruebas. Muestra que tanto para la generación completa como para la generación aleatoria, la configuración que incorpora información de tipo produce valores de cobertura más altos durante el tiempo de ejecución completo de los algoritmos de generación, en comparación con ignorar la información de tipo. Esto nuevamente respalda nuestra afirmación de que la información de tipo es beneficiosa cuando se generan pruebas unitarias para programas de Python. En general, por lo tanto, concluimos para nuestra segunda pregunta de investigación:

Resumen (RQ2): la incorporación de información de tipo es compatible con los algoritmos de generación de pruebas y les permite cubrir partes más grandes del código. Sin embargo, la fuerza de este efecto depende en gran medida del COU. Los proyectos que requieren tipos específicos de un gran grupo de tipos potenciales se benefician más y, por lo tanto, logran tamaños de efecto más grandes que los proyectos que solo utilizan tipos simples.

## 5 Trabajo relacionado

Lo más cercano a nuestro trabajo es la generación de pruebas de conjunto completo en EvoSuite [9] y la generación de pruebas aleatorias dirigidas por retroalimentación en Randoop [15]. Ambos enfoques apuntan a la generación de pruebas para Java, un lenguaje tipificado estáticamente, mientras que nuestro trabajo adapta estos enfoques a Python.

Hasta donde sabemos, se ha hecho poco en el área de generación de pruebas automatizadas para lenguajes tipeados dinámicamente. Enfoques como SymJS [12] o JSEFT [14] apuntan a propiedades específicas de las aplicaciones web de JavaScript, como el DOM del navegador o el sistema de eventos. Las pruebas aleatorias dirigidas por retroalimentación también se han adaptado a aplicaciones web con Artemis [3]. Un trabajo reciente propone LambdaTester [18], un generador de pruebas que aborda específicamente la generación de funciones de orden superior en lenguajes dinámicos. Nuestro enfoque, por el contrario, no se limita a dominios de aplicación específicos.

Para la generación automatizada de pruebas unitarias para Python, solo conocemos Auger8 ; genera casos de prueba a partir de ejecuciones SUT registradas, mientras que nuestro enfoque hace la generación automáticamente.

## 6. Conclusiones

En este documento, presentamos Pynguin, un marco de trabajo de generación de pruebas unitarias automatizadas para Python que está disponible como una herramienta de código abierto, y mostramos que Pynguin puede emitir pruebas unitarias para Python que cubren gran parte de las bases de código existentes. Pynguin proporciona un enfoque de generación de pruebas aleatorias y de conjunto completo, que evaluamos empíricamente en diez proyectos Python de código abierto. Nuestros resultados confirman hallazgos previos del mundo de Java de que un enfoque de conjunto completo puede superar a un enfoque aleatorio en términos de cobertura. Además, demostramos que la disponibilidad de información de tipo tiene un impacto en la calidad de generación de pruebas. Nuestras investigaciones revelaron una variedad de desafíos técnicos para las pruebas automatizadas

<sup>8</sup> <https://github.com/laffra/auger>, consultado el 25 de julio de 2020.

generación, que brindan amplias oportunidades para futuras investigaciones, por ejemplo, la integración de más algoritmos de generación de pruebas, como (Dyna)MOSA [16], la generación de afirmaciones o la integración de enfoques de inferencia de tipo.

## Referencias

1. Andrews, JH, Menzies, T., Li, FC: Algoritmos genéticos para unidades aleatorias pruebas. Trans. IEEE. Ing. de Software 37(1), 80–94 (2011)
2. Arcuri, A.: Realmente importa cómo se normaliza la distancia entre ramas en las pruebas de software basadas en búsqueda. suave Prueba. Verificación Reliab. 23(2), 119–147 (2013)
3. Artzi, S., Dolby, J., Jensen, SH, Møller, A., Tip, F.: Un marco para la prueba automatizada de aplicaciones web de JavaScript. En: Proc. CISE. págs. 571–580. MCA (2011)
4. Baresi, L., Miraz, M.: Testful: Generación automática de pruebas unitarias para clases Java. En: proc. CISE. vol. 2, págs. 281–284. MCA (2010)
5. Campos, J., Ge, Y., Albunian, N., Fraser, G., Eler, M., Arcuri, A.: Una evaluación empírica de algoritmos evolutivos para la generación de conjuntos de pruebas unitarias. información suave Tecnología 104, 207–235 (2018)
6. Csallner, C., Smaragdakis, Y.: Jcrasher: un probador automático de robustez para java. suave Practica Exp. 34(11), 1025–1050 (2004)
7. Fraser, G., Arcuri, A.: Evosuite: Generación automática de conjuntos de pruebas para orientación a objetos software. En: Proc. ESEC/FSE. págs. 416–419. MCA (2011)
8. Fraser, G., Arcuri, A.: La semilla es fuerte: estrategias de siembra en software basado en búsquedas pruebas. En: Proc. ICST. págs. 121–130. Comp. IEEE Soc. (2012)
9. Fraser, G., Arcuri, A.: Generación completa de conjuntos de pruebas. Trans. IEEE. Ing. de Software 39(2), 276–291 (2013)
10. Fraser, G., Arcuri, A.: Generación de prueba automatizada para genéricos de Java. En: Proc. SWQD. LNBPI, vol. 166, págs. 185–198. saltador (2014)
11. Levenshtein, VI: códigos binarios capaces de corregir eliminaciones, inserciones y reversiones. En: Doklady de la física soviética. vol. 10, págs. 707–710 (1966)
12. Li, G., Andreasen, E., Ghosh, I.: SymJS: prueba simbólica automática de aplicaciones web de JavaScript. En: Proc. FSE. págs. 449–459. MCA (2014)
13. Ma, L., Artho, C., Zhang, C., Sato, H., Gmeiner, J., Ramler, R.: Grt: prueba aleatoria guiada por análisis de programa (t). En: Proc. PLAZA BURSÁTIL NORTEAMERICANA. págs. 212–223. Comp. IEEE Soc. (2015)
14. Mirshokraie, S., Mesbah, A., Pattabiraman, K.: JSEFT: Generación automatizada de pruebas unitarias de javascript. En: Proc. ICST. págs. 1 a 10. Comp. IEEE Soc. (2015)
15. Pacheco, C., Lahiri, SK, Ernst, MD, Ball, T.: Prueba aleatoria dirigida por retroalimentación generación. En: Proc. CISE. págs. 75–84. Comp. IEEE Soc. (2007)
16. Panichella, A., Kifetew, FM, Tonella, P.: Generación de casos de prueba automatizada como un problema de optimización de muchos objetivos con selección dinámica de los objetivos. Trans. IEEE . Ing. de Software 44(2), 122–158 (2018)
17. Sakti, A., Pesant, G., Guéhéneuc, YG: Generador de instancias y representación de problemas para mejorar la cobertura del código orientado a objetos. Trans. IEEE. Ing. de Software 41(3), 294–313 (2014)
18. Selakovic, M., Pradel, M., Karim, R., Tip, F.: Generación de pruebas para funciones de orden superior en lenguajes dinámicos. proc. Programa ACM. Idioma 2 (OOPSLA), 161:1– 161:27 (2018)

<http://dx.doi.org/10.1037/0033-2909.101.1.19> Tonella, P.: Pruebas evolutivas de clases. En: Proc. ISSTA. págs. 100-1 119–128. MCA (2004)