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

EvoMaster - Mejoras de Usabilidad

Tesis de Licenciatura en Ciencias de la Computación

Philip Garrett

Director: Juan Pablo Galeotti

Buenos Aires, 2024

EVOMASTER - MEJORAS DE USABILIDAD

En el presente trabajo se introducen dos mejoras de usabilidad a EvoMaster, una herramienta open-source de generación automática de casos de test para APIs REST, GraphQL y RPC en base a algoritmos genéticos. El *core* de la herramienta es el encargado de utilizar algoritmos evolutivos, en particular el algoritmo de búsqueda MIO para generar los casos de test. Dicha generación abarca la caracterización del objeto de test y a su vez la escritura de los casos de test en el lenguaje elegido. Las mejoras se encuentran orientadas a la escritura de los casos. Por un lado se incorpora Python como una nueva elección de salida para los casos de *black-box* Fuzzing. Por el otro, se utiliza la información obtenida por el algoritmo genético para nombrar los casos de test en base a las acciones y objetos de test que el mismo se encuentre evaluando. Dichas mejoras se evaluaron utilizando APIs pertenecientes a un benchmark de EvoMaster.

Palabras claves: REST, GraphQL, RPC, APIs, Algoritmos Evolutivos, Test Naming, Python.

AGRADECIMIENTOS

En un lugar del Ática, de cuyo nombre no quiero acordarme, un costarricense nos respondió "Laif i' chourni"(Life is journey) a alguna pregunta que no guardaba relación con la respuesta. No obstante, la frase se convirtió en ícono del grupo. Hoy, permite hilvanar los agradecimientos y recordatorios a grupos y personas que me acompañaron durante estos años de estudio.

Por suerte en este viaje fui incorporando y no cambiando grupos de amigos. En un racconto en el que no promete ser breve, quiero recorrer momentos de este viaje a modo de agradecimiento.

Comienzo con aquel grupo de amigos en cual, terminado el secundario, cada uno comenzó su carrera y el descubrimiento de sus propios grupos. El tiempo nos encontraba cada 6 meses en juntadas hasta largas horas de la noche, al cabo de las cuales nos despedíamos por otros 6 meses. Recorrido el camino, nos encontramos ahora acompañando los proyectos de cada uno, compartiendo más seguido. Mucho ha sucedido en ese intermedio, un día me encontré caminando por la ciudad y el día soleado nos llevó a almorzar con una amiga al lado de la Embajada de Francia en una fiesta que celebraba dicha región.

Rememorando aquel viaje universitario que da inicio a este texto, me respondieron: "Dale, que falló el que siempre trae mate" a la invitación a tomar unos mates por fuera del aula. Así nació una amistad que nos permitió viajar y reír. Con el propósito de visitar a un amigo emigrado, y que años más tarde celebraría su matrimonio, viajamos a Europa. El CBC me presentó con grandes momentos, hubo los cuales en los que se discutía la regla de 3. Aquel histórico debate incluso quedó grabado en el yeso de algún desconocido. Dió también el puntapié inicial al grupo que se consolidó como "el de la facu", aquel con el que compartimos picadas en el espacio que hoy conocemos como el pabellón "0+infinito", partidos de ping-pong y fútbol 5 hasta las 12 de la noche y hasta viajes a esquiar. Ocasionalmente, hubo largas noches de trabajos prácticos y estudio, al fin y al cabo era el grupo de la facultad.

A la par, me encontraba tomando mates en Parque Las Heras, yendo a peñas, caminando a Luján o festejando casamientos. Muchas mudanzas ayudadas, incluso algunas a cargo de cuidar a un pequeño por unos instantes, con la promesa compartida de "si vos no lloras, yo no lloro". Grandes lugares de unión fueron aquellos, ya sea encontrándome con amigos paseando sus perros o haciendo degustación de vinos.

Si de plazas y parques hablamos, imposible olvidar que es lo único con lo que estaba en deuda aquel equipo que cuyo propósito inicial era jugar al rugby universitario. Por fortuna el tiempo nos encontró cada vez más lejos de la cancha y más cerca de una parrilla, cócteles, vinos y también casamientos. Nos dieron más risas.

En la segunda mitad de esta carrera universitaria, un nuevo trabajo me regaló el siguiente grupo de amigos. El viaje a la *meetup*, espontáneamente cantar *wimoweh* previo a una reunión, la reunión recurrente durante el 2020 para simular el espacio de oficina y acompañarnos mutuamente son algunos de los momentos coleccionados.

Cerca del final del camino y con el caballo un poco cansado, un grupo de gente joven pero con muchas ganas de estudiar me acompañó el último trecho. Tomamos a uno de los profesores como faro espiritual y nos ocupamos de que no queden dudas acerca de los ejercicios a ser evaluados. Dicho sea de paso, ¿quién dijo que no se puede apostar en un

laboratorio cuál va a ser el resultado de un trabajo práctico?

El pasado más reciente cuenta con muchos asados coordinados durante ratos de entrenamiento, mientras el cronómetro descuenta el tiempo del bloque y se escuchan anécdotas *allthenight*.

Gracias a todas las personas que formaron parte de este recorrido. Gracias por el aliento y acompañamiento, son varios quienes emigraron a Francia, España o Canadá pero siguieron de cerca. Pasaron distintos trabajos y momentos, cada uno con su aprendizaje. Gracias a la familia por este tiempo en la primera fila como espectadores.

Finalmente, gracias JP por el trabajo durante esta tesis. Fue un verdadero placer laburar en conjunto y hacer de la tesis algo divertido, que permite seguir aprendiendo. Especial gracias a vos y al DC con sus profesores. Me llevo 12 años de anécdotas.

Índice general

1. INTRODUCCIÓN

Parte fundamental del desarrollo de software es la escritura de casos de test que sirvan para poder verificar el comportamiento de un programa. Dichos tests cobran especial importancia al momento de incorporar funcionalidades a los sistemas, sin que se cambien las ya existentes. Llamamos una *suite* de tests al conjunto de tests que evalúan un sistema. La generación manual de los mismos puede ser tediosa, costosa e incompleta.

En las arquitecturas de microservicios, se ha vuelto muy popular el uso de APIs para exponer dichos microservicios. La utilización de las APIs permite una forma de comunicación del mundo exterior al servicio, definiendo por ejemplo los distintos puntos de acceso, los parámetros necesarios y las posibles respuestas. Existen APIs de tipo REST, GraphQL, RPC y SOAP.

EvoMaster [1] es una herramienta *open-source* para la generación automática de casos de test para los tres primeros tipos de API listados anteriormente. Utiliza el algoritmo evolutivo MIO [2] para dichos casos, buscando cubrir la mayor cantidad objetivos posibles: ya sea cubrimiento de líneas, *branches* o códigos de respuesta (*status codes*) en el contexto de APIs.

Los mejores resultados de EvoMaster se obtienen utilizando su configuración *white-box*, que permite la introspección e instrumentación del sistema a evaluar. Esta opción es la que ayuda a obtener mejores resultados en cuanto a métricas de cubrimiento de líneas o *branches* por ejemplo. No obstante, no siempre es posible contar con la instrumentación del sistema. EvoMaster se encuentra escrito en Java y proporciona *drivers* para la instrumentación en lenguajes de JVM y JavaScript. Por lo tanto, instrumentar sistemas escritos en lenguajes como Python o Go puede no ser posible. Es por ello que EvoMaster proporciona la opción *black-box* para la generación de casos de test. En estos casos, si de APIs REST hablamos, EvoMaster utilizará la especificación de la API para intentar cubrir la mayor cantidad de *endpoints* y *status codes* sin necesidad de instrumentar el sistema bajo evaluación (SUT por sus siglas en inglés).

Es en la opción *black-box* de EvoMaster que este trabajo incorpora la posibilidad de escribir el resultado de los tests en Python [10]. Proporcionando así al usuario un lenguaje más para la generación de sus casos. Python ofrece una curva de aprendizaje más amigable al usuario y para aquellos sistemas que no cuenten con la posibilidad de escribir un *driver* para su instrumentación, es posible que la escritura de casos de test en Python aporte mayor dinamismo y facilidad al momento de configurar sus ambientes de test, en comparación a entornos como Java o Kotlin.

A su vez, independiente de la modalidad de ejecución de EvoMaster, actualmente los casos de test generados son nombrados con un contador, provocando que en una misma clase, los casos de test sean llamados **test_1**, **test_2**, **test_3**, etc. Si pensamos en el código como una herramienta para que los usuarios puedan entender el desarrollo de un sistema, los objetivos que busca alcanzar y poder hacer referencia a los mismos, utilizar únicamente números como formato de nombre no es la mejor solución. Esta tesis busca extender la estrategia de nombrado, basándonos en el trabajo de Gordon Fraser et al. [5] para aportar semántica a los nombres de test, complementando con el trabajo de clusterización [4] de EvoMaster al momento de encontrar fallas en el sistema. Dado que EvoMaster utiliza un algoritmo genético, sabemos que contamos con los rasgos distintivos de cada test: qué

busca testear, cuál es la respuesta esperada y cómo se realizará la ejecución, por ejemplo.

La evaluación de las dos mejoras realizadas consistirá en verificar: la correcta ejecución de los casos de test escritos en Python y la unicidad de los nombres generados para identificar cada caso de test.

2. MARCO TEÓRICO

2.1. Search-based Testing

La escritura manual de casos de test puede ser tediosa e incluso incompleta. Puede que haya condiciones para las cuales sea necesaria una entrada difícil de reproducir manualmente, o que sencillamente no se nos haya ocurrido. Por otro lado, la generación aleatoria de tests también puede tener dificultades para generar casos de test que alcancen algunos puntos del código, tales como condiciones anidadas. Llevando también a *suites* de test incompletas.

La generación automática de casos de test puede ser pensada como un problema de optimización. En el mismo buscamos poder ejecutar la mayor cantidad de las líneas de código (*line coverage*) o evaluar todas las ramas de los condicionales del código (*branch coverage*). Decimos que estos son objetivos, *targets*, a cubrir dentro del programa que definiremos como espacio de búsqueda. En la Ingeniería del Software, estos espacios de búsqueda son grandes. Podemos pensar en cualquier software empresarial o masivo, que puede estar compuesto por miles de líneas de código y cientos de condicionales que afectan el flujo del programa, generando distintas opciones de resultados.

Search-based testing aplica algoritmos meta-heurísticos para resolver estos problemas de búsqueda, algunos de ellos son Hill Climbing o Tabu Search. En el caso de EvoMaster se utiliza el Algoritmo Genético MIO, explicado posteriormente, el cual se basa en generar una población de individuos (casos de test en este problema) e iterar para llegar a una solución óptima. La noción de Algoritmo Genético se inspira en la Teoría de la Evolución, la genética de un individuo y cómo la misma cambia de generación en generación. Se define una función de *fitness* que evalúa a cada individuo en relación a los objetivos que se buscan cubrir. Los mejores individuos se preservan y se reproducen a la siguiente generación mediante el *crossover* (cruza entre individuos) y la mutación para generar cambios. Dicho procedimiento se realiza hasta cubrir los objetivos o agotar el tiempo asignado a la generación de casos de test.

En el caso de EvoMaster, dado que el objeto de test son APIs, podemos pensar en el cubrimiento de los distintos *endpoints* provistos por una API o los distintos códigos de respuesta a sus operaciones.

2.2. Fuzzing

Fuzzing [3] es una técnica de *testing* que se basa en ejecutar el programa a evaluar utilizando datos de entrada inválidos o aleatorios. El objetivo es monitorear el comportamiento del programa en su ejecución con estos datos de entrada, buscando casos en los que el programa falle o incluso retorne resultados inválidos. Coloquialmente decimos que fuzzing estudia como el programa soporta "ruido" en el input.

Existen principalmente dos técnicas de fuzzing:

1. Black-box fuzzing

2. White-box fuzzing

Black-box fuzzing ejecuta sin conocer la estructura interna del programa a evaluar. Genera entonces datos aleatorios para probar, es por ello que una ejecución de testing aleatoria (Random Testing) es considerada black-box fuzzing. Si bien al no conocer la estructura interna del programa es más difícil lograr generar casos que evidencien comportamiento defectuoso de la aplicación, la ventaja de black-box fuzzing es que nos permite testear cualquier sistema y de manera fácil.

White-box fuzzing, en cambio, conoce el programa y utiliza eso para generar sus datos de prueba. Un white-box fuzzer podría utilizar las métricas de *line coverage* para saber que parte del programa ya fue testeado y así no gastar más tiempo en generar inputs que ataquen esa sección. Como contrapartida, white-box fuzzing es más complejo de implementar, ya que se debe poder acceder a algún tipo de especificación del programa que permita guiar al fuzzer.

EvoMaster implementa ambas técnicas de fuzzing para evaluar las APIs que se busquen testear. En el caso de APIs REST, por ejemplo, tan solo se necesita la ubicación de la especificación de la API en formato OpenAPI para que EvoMaster pueda ejecutar pruebas contra todos los *endpoints* y sus respectivos parámetros. Permitiendo así testear APIs cuyo código se encuentra escrito en otro lenguaje. White-box fuzzing en EvoMaster utiliza un controlador que se encarga de prender, apagar e incluso restaurar el estado del sistema bajo evaluación. Dicho controlador es configurado por quienes desarrollen el sistema, el mismo proporciona información sobre los paquetes a evaluar en el caso de aplicaciones basadas en lenguajes JVM y la especificación de la base de datos si hubiera una. Con esta información, EvoMaster es capaz de inspeccionar el sistema y generar inputs más precisos para testear las distintas condiciones que lo componen.

2.3. Algoritmo MIO

En la sección 2.1 se presentó la noción de Search-based Testing y se referenció al algoritmo MIO utilizado por EvoMaster. En la presente sección, se explicará en qué consiste el dicho algoritmo.

Nuevamente, si pensamos en la generación de casos de test como un problema de optimización, podemos rápidamente distinguir distintos objetivos de optimización: cubrimiento de ramas, líneas, códigos de respuesta, entre otros. Para alcanzar dichos objetivos, es importante tener en cuenta que:

- Los objetivos de test se pueden buscar independientemente. A su vez, se pueden incorporar nuevos tests sin necesidad de eliminar los ya existentes.
- Los objetivos pueden estar tanto relacionados (código anidado), como ser independientes (ubicados en distintas áreas del código).
- Algunos objetivos pueden ser imposibles de cubrir. Es posible que haya secciones del código que sean inaccesibles debido a los condicionales a ejecutar.
- El espacio de búsqueda puede ser muy grande, en especial para software no trivial en tests a nivel de sistema.

Teniendo en cuenta las características recién nombradas, se introduce el algoritmo MIO: *Many Independent Objective*. MIO busca mantener una población de n individuos por cada uno de los z objetivos a cubrir. La población inicial se genera aleatoriamente,

luego para las siguientes iteraciones se utiliza una probabilidad para decidir si se genera un nuevo test aleatoriamente o si se copia y muta uno de los ya existentes. El nuevo test es evaluado y dependiendo de su valor de *fitness* se guarda en 0 o más de las poblaciones existentes.

Para un objetivo k, si el mismo fue cubierto entonces su población decrece a 1, conteniendo el único test que lo cumple. Dicho test puede ser reemplazado si se genera:

- Un nuevo test que también cumpla todo el objetivo y sea más corto.
- Un test que sea de igual longitud, cubra la totalidad del objetivo y a su vez cubra mejor otros objetivos.

Si el objetivo no fue cubierto, se agrega el test. En caso de que el objetivo no cubierto ya cuente con la población máxima de casos de test, entonces el nuevo caso de test reemplaza aquel ya existente que tenga peor valor de *fitness*.

Para software no trivial, los objetivos de búsqueda pueden ser muchos, todos los distintos *endpoints* y *status codes* expuestos por una API por ejemplo. A su vez, los recursos destinados a la generación de tests pueden ser acotados. Por lo tanto, el algoritmo busca concentrarse en aquellos objetivos que sean más posibles de cumplir. Para ello, mantiene un contador por cada objetivo. Cada vez que se genera un nuevo caso de test en una población, se incrementa el contador. Luego, cada vez que se agrega un mejor test (o se reemplaza uno existente), el contador se reinicia. En las siguientes iteraciones, en lugar de evaluar una población cualquiera de las no cubiertas, se evalúa una de las que tenga el menor valor de contador. De esta manera, aquellos objetivos que mejoren serán elegidos más frecuentemente, mientras que aquellos que sean difíciles de cumplir y no mejoren no contarán con tanta inversión. Notar que una vez cubierto un objetivo, el mismo se elimina de la lista a evaluar, permitiendo concentrarse en aquellos objetivos no cumplidos.

A diferencia de otros algoritmos de búsqueda, MIO comienza en una etapa de búsqueda exploratoria para luego transicionar a una etapa de búsqueda enfocada. Esto se hace reduciendo linealmente los valores de la probabilidad de generar nuevos tests aleatoriamente y el valor máximo permitido de tests por objetivo, llevando a la probabilidad a 0 y el tamaño de población a 1. Al decrementar la población máxima permitida, cada vez que una objetivo contenga una población con un excedente de individuos, se elimina aquel que tenga peor valor de *fitness*.

A continuación, se presenta el pseudocódigo del algoritmo:

Algorithm 1 Algoritmo MIO

Input: Condición de parada C, Función de Fitness δ, Tamaño máximo población n, Probabilidad generación aleatoria P_r, Comienzo búsqueda enfocada F
Output: Set de individuos óptimos A

```
1: T ← setPoblacionVacia()
2: A ← {}
3: while ¬C do
4:     if P_r > rand() then
5:         p ← individuoAleatorio()
6:     else
7:         p ← individuoDeLaPoblacion(T)
8:         p ← mutar(p)
9:     end if
10:    for k ∈ objetivosAlcanzados(p) do
11:        if objetivoCubierto(k) then
12:            actualizarSet(A, p)
13:            T ← T \ {T_k}
14:        else
15:            T_k ← T_k ∪ {p}
16:            if |T_k| > n then
17:                eliminarPeorTest(T_k, δ)
18:            end if
19:        end if
20:    end for
21:    actualizarParametros(F, P_r, n)
22: end while
23: return A
```

2.4. Test Case Naming en EvoSuite

EvoSuite es una herramienta para la generación automática de casos de test para aplicaciones Java, concentrándose en objetivos tales como cubrimiento de ramas. Al igual que EvoMaster, utiliza un algoritmo genético para generar los casos de test.

EvoSuite, ya cuenta con un mecanismo para nombrar casos de test de forma descriptiva apoyándose en el cubrimiento obtenido por los casos de test generados [5]. En el mismo, se identifican tres requerimientos para obtener buenos nombres para los tests generados:

1. Los nombres deben ser descriptivos del código de test, con una relación intuitiva entre el código escrito y el nombre del test.
2. Cada nombre debe distinguir unívocamente a un test de otro en una *suite* de tests.
3. Los nombres deben permitir entender que parte del código fuente se está testeando sin tener que inspeccionar el código de test.

A su vez, identifican cuatro cubrimientos principales al momento de priorizar los objetivos a utilizar para nombrar los casos de test. En orden de prioridad:

1. Cubrimiento de excepciones: si un test genera una excepción, esta debe ser siempre informada en el nombre.

2. Cubrimiento de métodos: el nombre del test debe indicar cual es el método que se evalúa.

3. Cubrimiento de *output*: si hay varios tests para el mismo método, las distintas salidas del mismo aportan diferenciación.

4. Cubrimiento de *input*: aporta información respecto al ambiente de prueba.

El procedimiento entonces comienza identificando para cada test los objetivos alcanzados únicamente por cada uno. Una vez identificados, se obtienen los más importantes teniendo en cuenta la prioridad anteriormente descrita y se nombra al test con ellos. Luego, si hay tests con igual nombre, se resuelven las ambigüedades. Es posible que haya más de un caso de test que evalúa el mismo método, por ejemplo. En esos casos, para cada uno de los tests que comparten nombre, se obtiene la lista de objetivos unívocos en relación al subset de tests con nombre repetido. Es decir, se busca que es aquello que distingue a los casos de test, lo cual puede ser una condición distinta en ejecución por ejemplo. A continuación, para cada caso de test que no haya recibido un nombre, porque no es distinguido por un objetivo único por ejemplo, se busca el objetivo más importante que cubra y se lo utiliza como nombre. Finalmente, para todos los casos de test que tengan el mismo nombre si los hubiera, se agrega un sufijo numerando cada uno para aportar así un nombre único final.

A continuación, el pseudocódigo:

Algorithm 2 Algoritmo Generación de Nombres para Casos de Test Unitarios

Input: Suite de test T, Set de objetivos de cubrimiento G, Cantidad máxima de objetivos por nombre n

for $t \in T$ **do**
  $U \leftarrow objetivosCubiertos(\{t\}, G) \setminus objetivosCubiertos(T \setminus \{t\}, G)$
  $top \leftarrow objetivosPrincipales(U, n)$
  $nombre \leftarrow mergearTexto(top)$
  $nombrar(t, nombre)$
end for
for $T' \subset T$ tal que $\forall t \in T'$ tienen el mismo nombre **do**
  $resolverAmbigüedades(T')$
end for
for $t \in T$ **do**
  if t *no tiene nombre* **then**
    $C \leftarrow objetivosCubiertos(\{t\}, G)$
    $nombrar(t, head(objetivosPrincipales(C, n)))$
  end if
end for
for $t \in T$ **do**
  $resumirNombre(\{t\})$
end for
for $T' \subset T$ tal que $\forall t \in T'$ tienen el mismo nombre **do**
  $agregarSufijo(T')$
end for

2.5. Test Case Clustering en EvoMaster

La generación automática de casos de test puede resultar en test suites grandes que sean difíciles de investigar manualmente. Es necesario formalizar una taxonomía que permita a los desarrolladores identificar y concentrarse en las fallas más importantes. En este caso [4] se propone utilizar una *clusterización* basada en densidad para agrupar fallas. Llamaremos *clusters* a las clases de fallas propuestas por un método automatizado.

El trabajo de Andrea Arcuri et al utiliza EvoMaster para generar los casos de test y sobre ellos investiga las fallas encontradas. Tal como fue mencionado en las secciones anteriores, EvoMaster utiliza el algoritmo MIO para generar los casos. Dicho algoritmo busca maximizar los objetivos a cubrir, donde uno de ellos es el cubrimiento de líneas y otro las potenciales fallas del sistema bajo evaluación, por ejemplo.

Al momento de clasificar las fallas, es importante tener en cuenta las propiedades de *cohesión* y *separación* de los clusters de fallas propuestos por un mecanismo automatizado. Donde:

- *Separación* es el grado en el cual las fallas de una misma clase están presentes en el mismo *cluster*. Todo método automatizado debería poder garantizar que fallas similares se encuentran agrupadas en el mismo *cluster*.
- *Cohesión* responde al grado en el cual fallas de un mismo *cluster* son similares entre sí. Si un *cluster* contiene muchas fallas de distinta clase, pero las clases no están presentes en otros *clusters* (vale la *separación*), entonces es probable que el método de *clusterización* no esté siendo efectivo.

La taxonomía propuesta es para fallas en APIs de tipo REST. Los códigos de respuesta válidos para este tipo de APIs son: **1xx**, **2xx**, **3xx**, **4xx** y **5xx**. Identificando principalmente como fallas aquellas contenidas en la familia de los códigos **5xx**. Al evaluar un sistema sobre el cual se tiene control total, una respuesta 500 es muy probable que se deba a algún tipo de falla, como por ejemplo excepciones no controladas. A su vez, se consideran fallas a los resultados o respuestas de una API que no formen parte de su especificación de OpenAPI [7]. En detalle, la taxonomía principal propuesta es:

- Fallas de configuración o ejecución: responde a fallas al momento de configurar las herramientas utilizadas ya sea para realizar las pruebas o sistemas externos.
- Conflictos de especificación: diferencias entre el sistema bajo evaluación y la especificación OpenAPI provista.
- Fallas de integridad de datos: son aquellas relacionadas a la inyección de información para contar con datos de prueba, tanto en su inicialización como en la consistencia de los mismos.
- Fallas en la lógica de negocio: representan los problemas encontrados al momento de ejecutar las operaciones propias del sistema bajo evaluación.

En el presente trabajo, se utiliza la taxonomía propuesta para mejorar los nombres de los casos de test. Permitiendo por ejemplo el cambio de un nombre de caso de test como **test_1_with500** a **test_1_getOnItemsCauses500_internalServerError**.

3. PYTHON PARA BLACKBOX TESTING

En la presente sección se trata el primer objetivo de la tesis, en el cual se incorporó Python como salida a la generación de casos de test para la modalidad *black-box*.

3.1. Estado del Arte

Previo a la implementación realizada en este trabajo, EvoMaster contaba con la posibilidad de escribir los tests generados en los lenguajes Java y Kotlin para el framework JUnit en sus versiones 4 y 5, y JavaScript para su framework Jest.

Los lenguajes basados en la Java Virtual Machine (JVM) permiten generar tests de EvoMaster en sus opciones *white-box* y *black-box*, mientras que JavaScript solo para *black-box*. Si bien es cierto que los lenguajes de JVM permiten generar mejores casos de test al utilizar la opción *white-box*, también lo es que los mismos necesitan de mayor infraestructura para poder ser utilizados. Tanto Java como Kotlin son lenguajes compilados, lo cual involucra un ciclo de desarrollo un poco más largo para compilar y transformar a código objeto el propio código. A su vez, suelen necesitar ambientes de desarrollo que involucran un proyecto con gran manejo de dependencias. Mientras tanto, lenguajes interpretados como JavaScript y Python aportan mayor dinamismo y un manejo de dependencias más simple que no requiere de un gran proyecto para funcionar. En casos donde el usuario haya desarrollado su API en un lenguaje no JVM, poder contar con una opción dinámica para poder *testear* su desarrollo es de gran valor. Finalmente, en el estudio realizado por JetBrains [8], creadores del lenguaje Kotlin y de muchos ambientes de desarrollo ampliamente utilizados en la industria, acerca de los lenguajes de programación más populares al momento, se puede ver como tanto Python y JavaScript se encuentran en los lenguajes más usados. Lo cual refuerza la idea de que sean opciones de salida para EvoMaster.

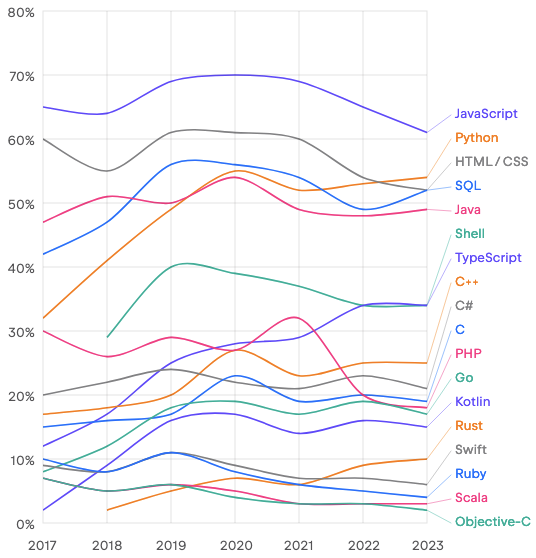

Fig. 3.1: Lenguajes más populares 2017-2023

3.2. Frameworks Utilizados

Para la escritura de los casos de test, se necesita la utilización de algún framework de test y otras bibliotecas que faciliten la resolución de tipos de datos u otras operaciones. En el caso del framework de test, las dos opciones más populares para Python son **Unittest** y **PyTest**. Se optó por utilizar **Unittest** por varios motivos. Principalmente, es el framework incluído en la biblioteca estándar de Python, lo cual permite garantizar que todo usuario del lenguaje va a poder hacer uso de sus funciones. Al ser parte de la biblioteca estándar cuenta con una gran comunidad y adopción en el ecosistema, lo cual facilita la resolución de errores que un usuario pueda encontrarse durante el desarrollo y testing de un sistema. A su vez, **Unittest** se encuentra inspirado en JUnit, framework utilizado para la escritura de casos de test Java y Kotlin en EvoMaster, permitiendo mantener un mismo estilo de escritura de tests. Finalmente, no hay un gran diferenciador de **PyTest** para el caso de uso de EvoMaster que presente una desventaja para **Unittest**. **PyTest** cuenta con más opciones de parametrización de los casos de test, pero esto no es un *feature* utilizado en los tests escritos por EvoMaster.

En cuanto a la lógica de los tests, se utilizan los módulos **requests**, **json**, **urllib3**, **rfc3986** y **timeout-decorator**.

3.2.1. Requests

Ejecución de Requests

La dependencia de **requests** [11] utilizada no es parte de la biblioteca estándar de Python, no obstante es de las más utilizadas y provee gran soporte para la realización de requests HTTP. Los métodos provistos llevan los nombres de los verbos HTTP, lo que permite una rápida y clara legibilidad de los requests GET, POST, etc. generados. Como es de esperar, se permite también la utilización de *headers* y *payloads*, así como la verificación de la respuesta, ya sea en su *status code* o *headers*.

Si bien no es parte de la biblioteca **requests**, se puede ver en los siguientes ejemplos la utilización de **timeout_decorator** que sirve para definir un tiempo de espera máximo para el test. Dicho tiempo se encuentra definido por EvoMaster en 60 segundos, al ser superado el tiempo de ejecución de un test en esa medida, el mismo falla. Esto previene que un test ejecute un request a un punto que tome demasiado tiempo en retornar y agregue entonces comportamiento errático o cause que la ejecución de los tests se extienda por mucho tiempo.

A continuación un ejemplo de un primer test ejecutando un request de tipo GET:

```
import requests
import timeout_decorator

class EvoMaster_successes_Test(unittest.TestCase):

    baseUrlOfSut = "http://localhost:8080"

    @timeout_decorator.timeout(60)
    def test_1(self):
        headers = {}
        headers['Accept'] = "application/json"
        res_0 = requests \
```

```
                .get(self.baseUrlOfSut + "/products",
                    headers=headers)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
```

Es posible también poder ejecutar otros requests, ya sean POST o DELETE por ejemplo. Así como también se permite que un mismo test realice más de un request en su ejecución, como se puede ver en el caso de **test_3**:

```
import requests
import timeout_decorator

class EvoMaster_successes_Test(unittest.TestCase):

    baseUrlOfSut = "http://localhost:8080"

    @timeout_decorator.timeout(60)
    def test_2(self):
        headers = {}
        headers["content-type"] = "application/x-www-form-urlencoded"
        body = "sourceFeature=x"
        headers['Accept'] = "*/*"
        res_0 = requests \
                .post(self.baseUrlOfSut + "/products/on7d/constraints/
                    excludes",
                    headers=headers, data=body)

        assert res_0.status_code == 201
        assert res_0.text == ''

    @timeout_decorator.timeout(60)
    def test_3(self):
        headers = {}
        headers['Accept'] = "application/json"
        res_0 = requests \
                .get(self.baseUrlOfSut + "/products",
                    headers=headers)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]

        headers = {}
        headers['Accept'] = "*/*"
        res_1 = requests \
                .delete(self.baseUrlOfSut + "/products/xX/constraints
                    /-1006003074",
                    headers=headers)

        assert res_1.status_code == 204
        assert res_1.text == ''
```

Respuestas JSON

JSON [9] es una herramienta muy utilizada para poder transmitir información entre sistemas de forma legible para las personas. Utiliza las estructuras de clave-valor y listas

para representar la información, de forma tal que operar su contenido se vuelve intuitivo para el usuario de un lenguaje de programación.

La biblioteca **requests** cuenta con la operación **json()** que retorna el contenido de la respuesta codificado en formato JSON. Dicha respuesta puede ser operada mediante los tipos de dato de Python como los mapas y las listas. De tal manera que se permita obtener la longitud de una respuesta, o acceder a un elemento de un objeto u array JSON. A continuación, se presentan dos ejemplos obtenidos de EvoMaster:

```
import unittest
import requests

class EvoMaster_successes_Test(unittest.TestCase):


    baseUrlOfSut = "http://localhost:8080"

    @timeout_decorator.timeout(60)
    def test_4(self):

        headers = {}
        headers['Accept'] = "application/json"
        res_0 = requests \
                .get(self.baseUrlOfSut + "/products",
                    headers=headers)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
        assert len(res_0.json()) == 30
        assert res_0.json()[0] == "ELEARNING_SITE"
        assert res_0.json()[1] == "h3yK"
        assert res_0.json()[2] == "j34TyA9a"

    @timeout_decorator.timeout(60)
    def test_5(self):

        headers = {}
        headers["content-type"] = "application/json"
        body = " { " + \
            " \"query\": \"  { remainder  (a : null)         } \" " + \
            " } "
        headers['Accept'] = "application/json"
        res_0 = requests \
                .post(self.baseUrlOfSut,
                    headers=headers, data=body)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
        assert res_0.json()["data"]["remainder"] == -1.0
```

3.2.2. Funciones de Utilidad

Es posible que en la ejecución de casos de test sean necesarias funciones de utilidad. En el caso de EvoMaster, las utilizadas por los tests en Python son *is_valid_uri_or_empty* y *resolve_location*.

Para la primera, se utiliza la biblioteca **rfc3986** y su función es la de verificar que una URI, por ejemplo provista por el header *Location* de una respuesta, sea válida. La función *resolve_location* utiliza la dependencia **urllib** con el fin de poder extraer la URI a la cual haya que realizar un request. Dicho valor se obtiene mediante un *Location* header y una URI parametrizada.

3.3. Modo Black-box

Tal como fue mencionado anteriormente, EvoMaster soporta las modalidades *white-box* y *black-box*. La diferencia principal entre ambas es que *white-box* lleva a cabo una instrumentación del código a testear, permitiendo obtener mejores resultados debido a poder contar con la introspección del código bajo test. Su contrapartida es poder disponer de un módulo que permita esa introspección, lo cual no es trivial para los distintos lenguajes de programación.

Por otro lado, *black-box* provee una opción que permite generar casos de test sin requerir un gran entorno. Continuando con la búsqueda de ambientes sencillos, se incluyó Python como opción para la generación de casos de test en modo *black-box*. De esta manera se cuenta con un lenguaje cuya curva de aprendizaje es amigable y permite testear los escenarios fundamentales de nuestra API.

Es importante tener en cuenta que en el modo *black-box*, soportan únicamente APIs de tipo REST y GraphQL.

3.4. Tokens y Cookies

Es posible que la API que buscamos testear requiera autenticación, es por eso que EvoMaster provee una forma de realizar *requests* de forma autenticada y poder llevar esa autenticación a los tests. Si la API requiere autenticación para ser utilizada, también la requerirá para ser testeada.

Para poder hacerlo, EvoMaster se apoya en la utilización y obtención de *tokens* y *cookies*. Dicho mecanismo ya se encuentra en uso para la escritura de tests en Java o Kotlin y dado que no es exclusivo a la modalidad *white-box*, debe ser soportado también en Python. Es importante tener en cuenta que en el presente trabajo no se modificó la forma de obtener las *cookies* o los *tokens*, sino que se incorporó su escritura en Python.

A continuación un ejemplo del resultado:

```
import unittest
import requests
from em_test_utils import *

class EvoMaster_faults_Test(unittest.TestCase):

    baseUrlOfSut = "http://localhost:5000"

    @timeout_decorator.timeout(60)
    def test_6(self):
        headers = {}
        headers["content-type"] = "application/json"
        body = " { " + \
            " \"username\": \"user\", " + \
            " \"password\": \"admin\" " + \
            " } "
```

```
        cookies_foo_jar = requests \
                .post(self.baseUrlOfSut + "/login",
                    headers=headers, data=body).cookies
        cookies_foo = requests.utils.dict_from_cookiejar(cookies_foo_jar)

        headers = {}
        headers['Accept'] = "application/json"
        res_0 = requests \
                .get(self.baseUrlOfSut + "/test",
                    headers=headers, cookies=cookies_foo)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
        assert res_0.json()["email"] == "foo@foo.foo"

    @timeout_decorator.timeout(60)
    def test_7(self):
        token_foo = "Bearer "
        headers = {}
        headers["content-type"] = "application/json"
        body = " { " + \
            " \"username\": \"user\", " + \
            " \"password\": \"admin\" " + \
            " } "
        res_foo = requests \
                .post(self.baseUrlOfSut + "/login",
                    headers=headers, data=body)
        token_foo = token_foo + res_foo.json()["token"]

        headers = {}
        headers["Authorization"] = token_foo # foo
        headers["content-type"] = "application/json"
        body = {}
        body = " { " + \
            " \"username\": \"user\", " + \
            " \"password\": \"admin\" " + \
            " } "
        headers['Accept'] = "application/json"
        res_0 = requests \
                .post(self.baseUrlOfSut + "/login",
                    headers=headers, data=body)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
        assert res_0.json()["token"] == "123"
```

3.5. REST vs GraphQL

En las secciones anteriores, los ejemplos provistos en Python siempre se trataron de APIs de tipo REST. Mostrando distintos tipos de request, la forma de corroborar el contenido de las respuestas y los códigos de respuesta.

Al momento de escribir un test GraphQL, el mismo guarda muchas similitudes con uno REST, con la diferencia de ser más acotado en sus opciones. Esto dado que los tests de GraphQL ejecutan un request HTTP con contenido JSON y esperan una respuesta del mismo tipo, utilizando el verbo HTTP Post. Por otro lado, una API REST podría recibir

un *request* de tipo GET o PUT, con su contenido en JSON o texto plano.

Otro punto interesante a tener en cuenta al momento de escribir tests para APIs de tipo GraphQL es que la API puede retornar un error y el mismo es parte del cuerpo de la respuesta, por lo tanto debe buscarse en la misma con un código de respuesta 200.

```
import unittest
import requests
from em_test_utils import *

class EvoMaster_successes_Test(unittest.TestCase):

    baseUrlOfSut = "http://localhost:8080/graphql"

    @timeout_decorator.timeout(60)
    def test_8(self):
        headers = {}
        headers["content-type"] = "application/json"
        body = " { " + \
            " \"query\": \"  { ordered4  (z : \\\"szay9finEYMY4CP\\\")
                        } \" " + \
            " } "
        headers['Accept'] = "application/json"
        res_0 = requests \
                .post(self.baseUrlOfSut,
                    headers=headers, data=body)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
        assert len(res_0.json()["errors"]) == 1
        assert res_0.json()["errors"][0]["message"] == "Internal Server
            Error(s) while executing query"
        assert len(res_0.json()["errors"][0]["locations"]) == 0
        assert res_0.json()["data"]["ordered4"] is None


    @timeout_decorator.timeout(60)
    def test_9(self):
        headers = {}
        headers["content-type"] = "application/json"
        body = " { " + \
            " \"query\": \"  { costfuns  (i : null,s : \\\"F\\\")         }
                \" " + \
            " } "
        headers['Accept'] = "application/json"
        res_0 = requests \
                .post(self.baseUrlOfSut,
                    headers=headers, data=body)

        assert res_0.status_code == 200
        assert "application/json" in res_0.headers["content-type"]
        assert res_0.json()["data"]["costfuns"] == "10"
```

Tal como se puede ver en el código provisto, el **test_8** muestra un error que se obtiene de buscar en la respuesta el primero error de un arreglo de errores, mientras que el campo *data* no provee información. En cambio, en **test_9**, se puede ver como se busca el valor *costfuns* especificado en la *query* dentro del campo *data*, sin que haya algún error.

3.6. Evaluando los tests

Al momento de evaluar la solución, se utilizaron APIs pertenecientes a EMB [6]. Las mismas incluyen APIs puramente de test generadas para casos artificiales y otras ya existentes públicamente en GitHub. Contamos entonces con:

- 6 APIs de tipo REST
- 3 APIs de tipo GraphQL

Para cada una de ellas, se ejecutó EvoMaster con un presupuesto de tiempo máximo de 10 minutos y luego los tests generados fueron ejecutados. La cantidad total de tests generados por API fue:

Tipo API	Nombre API	#tests
REST	catwatch	45
REST	features-service	32
REST	ncs	15
REST	scs	13
REST	news	17
REST	session-service	22
GraphQL	ncs	6
GraphQL	scs	11
GraphQL	petclinic	16

Tab. 3.1: Cantidad de tests Python generados por API

Generando entonces un total de 177 tests, de los cuales:

- 144 tests para APIs REST
- 33 tests para APIs GraphQL

Al ejecutar cada uno de esos tests, el resultado final fue:

Exitosos	165
Error de Aserción	9
Tiempo Excedido	3
Error de compilación	0

Tab. 3.2: Ejecución tests Python

Si bien no es esperado que tests recientemente generados fallen con errores de aserción o tiempo excedido, dichas fallas corresponden a la lógica de negocio de la API bajo evaluación. En todos los casos, el programa de test fue capaz de ejecutar sin errores de compilación, el cual es el verdadero objetivo de la implementación realizada.

4. ESTRATEGIAS PARA NOMBRADO DE TESTS

En esta sección se presenta el segundo objetivo de la tesis donde se busca obtener un algoritmo para nombrar los casos de test generados utilizando información de los mismos.

4.1. Estado del Arte

Al momento de iniciar este trabajo, EvoMaster no contaba con opciones para nombrar los casos de test generados. Los mismos se nombraban bajo el prefijo **test_** y un número autoincremental. En caso de que el test fuese de una API REST o GraphQL los tests se ordenaban bajo algún criterio y aquellos en los que se retornaba algún código de respuesta 500, se incorporaba además el sufijo **_with500** al nombre. De fallar esta incorporación o el ordenamiento, se nombraba únicamente con un número, como estrategia de *fallback*. A continuación el pseudocódigo:

Algorithm 3 Denominación Legacy

Input: Individuos *inds*, Organizador Test Suite *sorter*
Output: Set de Casos de Test T

```
1: if algun(inds) is RPCIndividual then
2:     i ← 0
3:     while i ≠ len(inds) do
4:         nombre ← "test_$i"
5:         T.agregar(TestCase(inds[i], nombre)
6:         i ← i + 1
7:     end while
8: else
9:     res ← sorter.nombrar(inds)
10:    if huboError(res) then
11:        i ← 0
12:        while i ≠ len(inds) do
13:            nombre ← "test_$i"
14:            T.agregar(TestCase(inds[i], nombre)
15:            i ← i + 1
16:        end while
17:    else
18:        T ← res
19:    end if
20: end if
21: return T
```

A su vez, tal como fue presentado en el Marco Teórico, EvoMaster cuenta con una estrategia de clusterización para casos de test. La misma agrupa los casos de test generados en distintas *suites*. Las categorías finales en las que se generan los casos de test son:

- *successes*: Contiene los casos de test que contienen llamadas exitosas.

- *faults*: Contiene los casos de test que posiblemente indiquen fallas.
- *fault_representatives*: Contiene un caso de test por falla encontrada. Los casos de test en este archivo son un subset de los casos de test que puedan indicar fallas.
- *others*: Contiene los casos de test que involucran llamadas con error, pero no representan fallas.

4.2. Algoritmo de Denominación

Dado que EvoMaster utiliza un algoritmo genético para la generación de los casos de test, se cuenta con las distintas acciones que componen a un caso de test, así como su genética. Dichas características son las que permiten nombrar a los casos de test con sus puntos más relevantes.

Un punto importante al buscar nombrar los casos de test, es que se intenta obtener nombres distintos para cada uno. Por lo tanto, una vez generados los nombres, se lleva a cabo un proceso de desambiguación que varía para cada tipo de API bajo test. Dicho procedimiento es explicado más adelante.

4.2.1. Los Últimos Serán los Primeros

Cada caso de test puede estar formado por múltiples acciones. Por ejemplo, un test puede primero buscar información de un ítem para luego modificar su descripción. En dicho test contaremos entonces con dos acciones, una para buscar y la siguiente para modificar.

Dado que la ejecución de EvoMaster se basa en *search-based testing*, podemos asumir que si contamos con un test como el del ejemplo previo, vamos a contar también con uno que únicamente realiza la búsqueda de la información del ítem. Esto porque para poder buscar esa información, necesitamos conocer el *endpoint*, lo cual lo transforma en un objetivo de búsqueda para EvoMaster, por ende en un objetivo para testear. Esto es de suma importancia dado que nos permite pensar en que la última acción de un test sea aquella más relevante para el contexto de lo que se quiere testear, y por ende es la que utilizaremos para nombrar test.

A su vez, existen distintos modificadores que agregan información al nombre del test. Algunos casos de test pueden necesitar de una base de datos para la configuración de los mismos, o incluso de objetos que simulen un servicio externo. En esos casos, se incluye un sufijo indicando la presencia de estos servicios, los cuales son:

- SQL
- Mongo
- WireMock

Finalmente, retomando el concepto de Clusterización ya presentado, EvoMaster categoriza fallas que puedan formar parte de un caso de test. Aquellas lo hagan son incorporadas al nombre del test mediante una descripción ya definida. Algunos ejemplos de las mismas son:

Nombre	Descripción
HTTP status 500	causes500_internalServerError
Sintaxis de respuesta inválida	rejectedWithInvalidPayloadSyntax
Respuesta no válida de acuerdo a la especificación	returnsSchemaInvalidResponse
Acceso no autorizado de un recurso protegido	allowsUnauthorizedAccessToProtectedResource

Tab. 4.1: EvoMaster *FaultCategory* descripciones

4.2.2. Denominación por Tipo

Como ya sabemos, EvoMaster soporta evaluar distintos tipos de APIs. Cada uno de esos tipos cuenta con particularidades. Por ejemplo, todo *request* a una API GraphQL es de tipo POST, mientras que una API REST puede recibir requests de todos los demás verbos HTTP (GET, PUT, DELETE, etc). Es por ello que cada API debe nombrar sus tests de forma particular.

En esta sección, presentaremos el pseudocódigo para cada tipo de API, bajo la función *expandName*, la cual será referenciada luego por el algoritmo a nivel general.

REST

En el contexto de APIs REST, definimos los siguientes puntos como importantes, en el orden:

1. El verbo HTTP del request.
2. La ruta del servicio a la que se ejecuta el request.
3. El resultado o respuesta del servicio.

Una vez incorporado el verbo se toma la última parte de la ruta. No se utiliza la ruta completa ya que potencialmente pueden ser muy largos para una ruta muy compleja. En caso de que la última sección sea un parámetro, se toma su antecesor. Por ejemplo:

- */users/*{*id*}*/balance*: *balance*
- */users/*{*id*}: *user*
- */users*: *users*

Finalmente, para el resultado obtenido, si la llamada fue de tipo GET y la respuesta es de código 200 entonces se genera información de acuerdo a la respuesta. En el caso de que la respuesta sea de tipo JSON y el valor un arreglo vacío se toma el sufijo *returnsEmptyList*. De forma similar se analizan los casos de listas con uno o más elementos, objetos vacíos o con contenido o simplemente texto plano. Para aquellos casos en los que la llamada no es de tipo GET o la respuesta no fue de código 200 entonces se agrega el código de respuesta devuelto por el servicio.

La función de denominación para una API de tipo REST es entonces:

Algorithm 4 expandName

Input: Individuo *ind*, Tokens de denominación *nameTokens*
Output: Nombre del test *nombreFinal*

ultimaAccion ← *ind.acciones.ultima*
nameTokens.agregar(*ultimaAccion.verbo*, *'on'*, *ultimaAccion.calificadorRuta*)
resultadoAccion ← *ultimaAccion.resultado*
nameTokens.agregar(*'returns'*)
if *esDeTipoGet*(*ultimaAccion*) && *resultadoAccion.codigo* == 200 **then**
    if *resultadoAccion.bodyesVacio* **then**
        nameTokens.agregar(*'empty'*)
    else
        if *esDeTipoJson*(*resultadoAccion*) **then**
            nameTokens.agregar(*parseJson*(*resultadoAccion*))
        else
            nameTokens.agregar(*'content'*)
        end if
    end if
else
    nameTokens.agregar(*resultadoAccion.codigo*)
end if
nombreFinal ← *formatearNombre*(*nameTokens*)
return *nombreFinal*

La función *parseJson* se asume que respeta el formato descrito anteriormente con el ejemplo de la lista vacía. Por otro lado, la función *formatearNombre* será presentada en las secciones siguientes.

Utilizando el lenguaje Java como tipo de salida, el siguiente test es un ejemplo de la estrategia para nombrar casos de test REST:

```
@Test(timeout = 60000)
public void test_0_getOnStatisticsReturnsEmptyList() throws Exception {

    given().accept("application/json")
            .get(baseUrlOfSut + "/statistics")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("application/json")
            .body("size()", equalTo(0));
}
```

GraphQL

En el caso de GraphQL, el procedimiento es similar, pero sus opciones son más acotadas. Definimos los siguientes puntos como importantes, en el orden:

1. El tipo de método a ejecutar: *query* o *mutation*
2. El método sobre el cual se ejecuta
3. La respuesta del servicio.

Los primeros dos puntos no varían en su forma con respecto a REST. En cuanto a la respuesta, de ser exitosa la operación la información es devuelta en el campo *data* de la respuesta, mientras que los errores en el campo *errors*. Tal como muestra el siguiente ejemplo:

```
{
  "data": {},
  "errors": [
    {
      "message": "This operation failed"
    }
  ]
}
```

Por ende, al incorporar al nombre el valor de la respuesta, se debe inspeccionar la respuesta para ver si la misma contiene errores, data o está vacía.

Algorithm 5 expandName

Input: Individuo *ind*, Tokens de denominación *nameTokens*
Output: Nombre del test *nombreFinal*

1: $ultimaAccion \leftarrow ind.acciones.ultima$
2: $nameTokens.agregar(ultimaAccion.tipoMetodo, 'on', ultimaAccion.nombreMetodo)$
3: $resultadoAccion \leftarrow ultimaAccion.resultado$
4: $nameTokens.agregar('returns')$
5: **if** $resultadoAccion.contieneErrores()$ **then**
6: $\quad nameTokens.agregar('error')$
7: **else**
8: $\quad$ **if** $resultadoAccion.tieneDataNoVacia()$ **then**
9: $\quad\quad nameTokens.agregar('data')$
10: $\quad$ **else**
11: $\quad\quad nameTokens.agregar('empty')$
12: $\quad$ **end if**
13: **end if**
14: $nombreFinal \leftarrow formatearNombre(nameTokens)$
15: **return** $nombreFinal$

Generando así tests como el siguiente:

```
@Test(timeout = 60000)
public void test_1_queryOnRegexReturnsData() throws Exception {
    given().accept("application/json")
            .contentType("application/json")
            .body(" { " +
                " \"query\": \"  { regex  (txt : \\\"Ld5qStmfNzdNBjl\\\")
                            } \" " +
                " } ")
            .post(baseUrlOfSut)
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("application/json")
            .body("'data'.'regex'", containsString("none"));
}
```

RPC

Finalmente, para RPC los nombres toman en cuenta:

1. El nombre de la clase a ejecutar
2. El nombre del método o función a ejecutar
3. La respuesta del servicio.

Al evaluar la respuesta, la misma puede haber generado una falla, por lo tanto en ese caso se incluye el nombre de la excepción generada. En caso contrario, se incluye únicamente si la llamada fue exitosa o no sin importar el valor generado. Su pseudocódigo entonces es:

Algorithm 6 expandName

Input: Individuo *ind*, Tokens de denominación *nameTokens*
Output: Nombre del test *nombreFinal*

ultimaAccion ← *ind.acciones.ultima*
nameTokens.agregar(*ultimaAccion.nombreClase*,′ *on*′, *ultimaAccion.nombreMetodo*)
resultadoAccion ← *ultimaAccion.resultado*
if *resultadoAccion.contieneFallas*() **then**
    nameTokens.agregar(′*throws*′)
    nameTokens.agregar(*resultadoAccion.obtenerExcepcion*())
else
    nameTokens.agregar(′*returns*′)
    if *resultadoAccion.fueExitosa*() **then**
        nameTokens.agregar(′*success*′)
    else
        nameTokens.agregar(′*error*′)
    end if
end if
nombreFinal ← *formatearNombre*(*nameTokens*)
return *nombreFinal*

A continuación un ejemplo de test RPC:

```
@Test(timeout = 60000)
public void
    test_2_ncsServiceBlockingStubOnGammqThrowsStatusRuntimeException()
    throws Exception {
    try{
        org.grpc.ncs.generated.DtoResponse res_0 = null;
        {
            org.grpc.ncs.generated.GammqRequest arg0 = null;
            res_0 = var_client0_NcsServiceGrpc_NcsServiceBlockingStub.gammq
                (arg0);
        }// org/grpc/ncs/imp/Gammq_80_exe UNEXPECTED_EXCEPTION:io.grpc.
            StatusRuntimeException
    } catch(Exception e){ /* UNKNOWN */ }
}
```

4.2.3. Desambiguando REST

En el caso de APIs REST, se optó por dar un paso más en la denominación de los casos de test. Este paso implica el análisis de los nombres generados para encontrar aquellos que tengan el mismo. Para dichos casos de test se invoca una función cuyo objetivo es desambiguar los nombres.

Esta función recibe los individuos que comparten nombre y cuenta con dos formas de desambiguar: tomar la ruta sobre la cual se ejecutan los requests o inspeccionar si alguno de los requests utiliza *query params*.

Ruta

Para el caso de la ruta entonces, para desambiguar los casos de test de una API REST se busca aquellos que compartan el último valor de la ruta elegida anteriormente y se agrega el segmento anterior. A modo de ejemplo, sean las siguientes rutas de dos tests que realizan un GET:

- */my/funny/path*
- */my/funniest/path*

Donde sus respectivos tests tienen el nombre:

- *test_0_getOnPath*
- *test_1_getOnPath*

El algoritmo en este caso incorporará el segmento anterior generando los nombres:

- *test_0_getOnFunnyPath*
- *test_1_getOnFunniestPath*

Query Param

En el caso de los *query params*, se verifica si hay alguno que los utilice. Por ejemplo en los siguientes casos que realizan un GET:

- */syntax/languages*
- */syntax/languages?myQueryParam=value*

El algoritmo en este caso incorporará el modificador *withQueryParam* dando así los tests:

- *test_0_getOnSyntaxLanguagesReturnsEmpty*
- *test_1_getOnLanguagesWithQueryParamReturnsEmpty*

De haber más de un parámetro en la llamada, se utiliza el plural *withQueryParams*.

Algoritmo modificado

Para poder realizar esto, se modificó la firma del método *expandName* presentado en la sección anterior para que incluya una función de desambiguación, transformando la función de la siguiente manera:

Algorithm 7 expandName

Input: Individuo *ind*, Tokens de denominación $nameTokens$, Función de desambiguación $ambiguitySolver$
Output: Nombre del test $nombreFinal$

```
1: ultimaAccion ← ind.acciones.ultima
2: nameTokens.agregar(ultimaAccion.verbo, 'on')
3: if ambiguitySolver == null then
4:     nameTokens.agregar(ultimaAccion.calificadorRuta)
5: else
6:     nameTokens.agregar(ambiguitySolver(ultimaAccion))
7: end if
8: // Continua con la ejecución como antes
9: nombreFinal ← formatearNombre(nameTokens)
10: return nombreFinal
```

Cabe destacar que la ejecución de tanto GraphQL como RPC cuenta con el mecanismo para desambiguar, pero el mismo se encuentra en fase experimental, como será mencionado luego.

4.2.4. Respetando Convenciones

Una parte importante de generar nombres para casos de test es respetar las convenciones de cada lenguaje. Como fue mencionado anteriormente, EvoMaster cuenta con la posibilidad de generar sus salidas tanto en Java, Kotlin, JavaScript y a partir de este trabajo Python.

Cada uno de esos lenguajes cuenta con convenciones distintas para los nombres de sus métodos:

- Java y Kotlin suelen utilizar camelCase
- Python suele utilizar snake_case
- JavaScript suele utilizar PascalCase

Por lo tanto, la función *formatearNombre* utilizada por los pseudocódigos de *expandName* garantiza este comportamiento:

Algorithm 8 formatearNombre

Input: Tokens de denominación $nameTokens$, Formato de salida $outputFormat$
Output: Nombre del test $nombreFinal$

if $outputFormat.isJavaOrKotlin()$ **then**
  $nombreFinal \leftarrow formatCamelCase(nameTokens)$
else
  if $outputFormat.isPython()$ **then**
    $nombreFinal \leftarrow formatSnakeCase(nameTokens)$
  else
    if $outputFormat.isJavaScript()$ **then**
      $nombreFinal \leftarrow formatPascal(nameTokens)$
    else
      $throwerror(outputFormatnoreconocido)$
    end if
  end if
end if
return $nombreFinal$

4.2.5. Algoritmo Obtenido

Tomando los pseudocódigos presentados anteriormente para las funciones *expandName* y *formatearNombre*, presentamos el pseudocódigo del algoritmo de denominación.

Algorithm 9 Algoritmo Generación de Nombres para Casos de Test de APIs

Input: Set Individuos I, Formato de salida $outputFormat$

$nombrePorIndividuo \leftarrow mapaVacio()$
for $ind \in I$ **do**
  $nombrePorIndividuo[ind] \leftarrow expandName(ind, listaVacia(), null)$
end for
for $T' \subset T$ tal que $\forall t \in T'$ tienen el mismo nombre **do**
  $ambiguitySolver \leftarrow obtenerFuncionDesambiguadora(T')$
  while $hayaNombresParaDesambiguar(T')$ **do**
    for $ind \in T'$ **do**
      $nombrePorIndividuo[ind] \leftarrow expandName(ind, listaVacia(), ambiguitySolver)$
    end for
  end while
end for
for $< ind, nombre > \in nombrePorIndividuo$ **do**
  $nombrePorIndividuo[ind] \leftarrow agregarPrefijo(nombre)$
end for
return $nombrePorIndividuo$

Donde *hayaNombresParaDesambiguar* verifica que no haya más casos de test para desambiguar, ya sea que todos fueron procesados o no hay más formas de desambiguar los restantes. Por su parte, la función *agregarPrefijo* simplemente incorpora el String *test_$counter* donde *counter* es un autoincremental que diferencia todos los tests en un test suite.

4.3. Evaluando los Resultados

Con respecto a la evaluación de la estrategia de denominación, la misma se divide en dos partes:

1. Resultados para APIs GraphQL y RPC
2. Resultados para APIs REST

La razón para esta división es que tanto las APIs RPC como GraphQL no cuentan con una estrategia para desambiguar los nombres repetidos.

Todas las ejecuciones realizadas en esta evaluación fueron realizadas eligiendo **JAVA_JUNIT_4** como formato de salida y una ejecución de EvoMaster de 10 minutos de duración. A su vez, dado que EvoMaster puede generar distinta cantidad de tests entre ejecuciones, la mayoría de las comparaciones a realizar serán en base a porcentajes y no cantidad de tests, aunque la misma sea mencionada.

4.3.1. GraphQL y RPC

Al no haber forma de desambiguar los nombres, lo presentado aquí se basará en comparar para cada tipo de API los resultados obtenidos.

GraphQL

Este tipo de APIs soporta ejecución tanto en modo *black-box* como *white-box*. Dado que *white-box* lleva a cabo una exploración del código objetivo más exhaustiva, se espera que haya una mayor cantidad de tests generados para dicho modo.

Más aún, al realizar introspección del código, es esperable que haya más de un test con el mismo nombre. Dado que estos tests pueden ejecutar distintas ramas de la función bajo evaluación, llevando así a testear el mismo método con distintos parámetros y obteniendo entonces tests distintos pero con el mismo nombre.

	Black-box		White-box	
	#tests	% nombres repetidos	#tests	% nombres repetidos
SCS	11	0	32	0
NCS	6	0	13	31
Petclinic	16	0	32	41

Tab. 4.2: Tests repetidos GraphQL

Tal como era esperado, la ejecución *white-box* es la que tiene nombres duplicados. Es posible que la SCS no tenga nombres repetidos en *white-box* debido a un espacio de búsqueda más acotado o con poca variabilidad.

A continuación, se presentan dos ejemplos de nombres en distintos tests:

```
@Test(timeout = 60000)
public void test_0_queryOnExpintReturnsError() throws Exception {

    given().accept("application/json")
            .contentType("application/json")
            .body(" { " +
                " \"query\": \"  { expint           } \" " +
                " } ")
            .post(baseUrlOfSut + "/graphql")
            .then()
            .statusCode(200)// org/graphqlncs/type/Expint_26_exe
            .assertThat()
            .contentType("application/json")
            .body("'errors'.size()", equalTo(1))
            .body("'errors'[0].'message'", containsString("Internal Server
               Error(s) while executing query"))
            .body("'errors'[0].'locations'.size()", equalTo(0))
            .body("'data'.'expint'", nullValue());
}

@Test(timeout = 60000)
public void test_1_mutationOnUpdateSpecialtyReturnsDataUsingSql() throws
    Exception {
    List<InsertionDto> insertions = sql().insertInto("owners", 522L)
            .d("first_name", "\"tdD35ZI1Ut\"")
            .d("last_name", "\"_EM_1555_XYZ_\"")
            .d("address", "NULL")
            .d("city", "\"ULuRLQNS\"")
            .d("telephone", "\"YeD1NtiPL9\"")
        .dtos();
    InsertionResultsDto insertionsresult = controller.
       execInsertionsIntoDatabase(insertions);

    given().accept("application/json")
            .contentType("application/json")
            .body(" { " +
                " \"query\": \"mutation{    updateSpecialty  ( input:{
                   specialtyId:1, name:\\\"h\\\"})    {specialty{name}}
                      }\" " +
                " } ")
            .post(baseUrlOfSut + "/graphql")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("application/json")
            .body("'data'.'updateSpecialty'.'specialty'.'name'",
               containsString("h"));
}
```

RPC

De forma similar evaluamos RPC, con la diferencia de que este tipo de APIs no puede ser ejecutada en modo *black-box*. Por lo tanto se cuenta únicamente con ejecuciones *white-box* para las distintas APIs evaluadas:

Dado que RPC presenta un espacio de búsqueda acotado, con pocos tests generados,

	#tests	% nombres repetidos
grpc-ncs	11	0
grpc-scs	22	0
thrift-ncs	11	0
thrift-scs	13	15

Tab. 4.3: Tests repetidos RPC

es esperable que haya pocos repetidos.

Generando así, tests como los siguientes:

```
@Test(timeout = 60000)
public void test_0_scsServiceBlockingStubOnTitleThrowsRuntimeException()
    throws Exception {
    try{
        org.grpc.scs.generated.DtoResponse res_0 = null;
        {
         org.grpc.scs.generated.TitleRequest arg0 = null;
         {
          org.grpc.scs.generated.TitleRequest.Builder arg0builder = org.
              grpc.scs.generated.TitleRequest.newBuilder();
          arg0builder.setSex("Bmj0V1pDjGuu8");
          arg0builder.setTitle(null);
          arg0 = arg0builder.build();
         }
         res_0 = var_client0_ScsServiceGrpc_ScsServiceBlockingStub.title(
             arg0);
        }// RPC_framework_code UNEXPECTED_EXCEPTION:java.lang.
            RuntimeException
    } catch(Exception e){
        // ERROR: fail to instance value of input parameters based on dto/
            schema, msg error:fail to find the builder:org.grpc.scs.
            generated.TitleRequest with error msg:null
    }
}

@Test(timeout = 60000)
public void test_1_scsServiceBlockingStubOnCookieReturnsSuccess() throws
    Exception {
    org.grpc.scs.generated.DtoResponse res_0 = null;
    {
     org.grpc.scs.generated.CookieRequest arg0 = null;
     res_0 = var_client0_ScsServiceGrpc_ScsServiceBlockingStub.cookie(arg0)
         ;
    }
    assertEquals("0", res_0.getValue());
}
```

4.3.2. REST

Al evaluar REST, lo haremos comparando las siguientes 6 APIs:

- catwatch
- scs

- session-service
- features-service
- news
- ncs

Para cada una de ellas, se compara el porcentaje de tests con nombre repetido para *black-box* (bb) y *white-box* (wb). Tal como sucedió para el caso de las APIs GraphQL, se espera que el porcentaje de tests con nombre repetido sea mayo para el caso de ejecuciones *white-box*, por los mismos motivos que GraphQL.

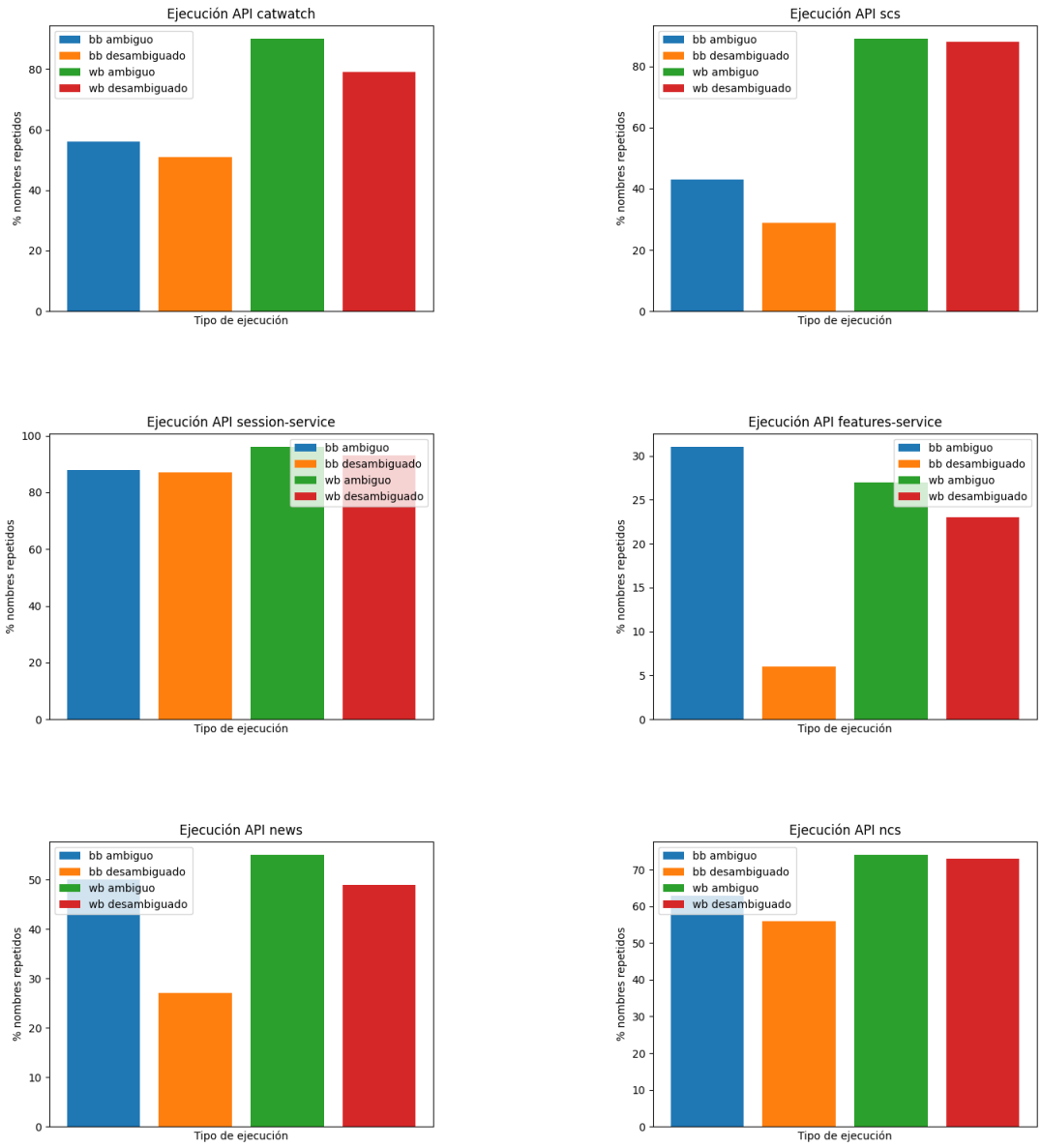

Fig. 4.1: Porcentaje de nombres repetidos por tipo de ejecución y API

Al mirar los gráficos presentados, vemos que la hipótesis se comprueba para la gran mayoría de los casos, teniendo como excepción a la API *features-service*. Al mirar la cantidad de tests generados en esta API, vemos la siguiente comparación:

	Ambiguo		Desambiguado	
	Black-box	White-box	Black-box	White-box
#tests	32	22	32	26

Tab. 4.4: Tests generados *features-service*

Nótese como la ejecución *white-box* genera menos tests en ambos casos. Es decir que son necesarios menos tests para cubrir el espacio de búsqueda, al haber menos tests generados hay menos tests sobre el mismo *endpoint*, lo cual es consistente con lo informado mediante los gráficos.

Mejor Caso Black-Box

Continuamos con la evaluación de la API *features-service* para comparar los distintos nombres de los casos de test generados. Veamos por ejemplo los siguientes casos de la suite de tests *EvoMaster_successes_Test.java*:

- test_0_getOnProductFeaturesReturnsEmptyList
- test_1_getOnConfigurFeaturesReturnsEmptyList
- test_2_getOnConfigurationsReturnsEmptyList
- test_3_getOnProductsReturns183Elements
- test_4_getOnProductReturnsObject
- test_5_getOnConfigurReturnsObject
- test_6_getOnSwagger_jsonReturnsObject
- test_7_putOnFeaturReturns200
- test_8_postOnProductReturns201
- test_9_postOnConfigurReturns201
- test_10_postOnRequiresReturns201
- test_11_postOnExcludesReturns201
- test_12_postOnFeaturReturns201
- test_13_deleteOnFeaturReturns204
- test_14_deleteOnConfigurReturns204
- test_15_getOnConfigurReturns204
- test_16_deleteOnProductReturns204

- test_17_deleteOnConstraintReturns204
- test_18_deleteOnConstraintReturns204

Se puede ver como se aprovechan los recursos descritos anteriormente, tales como diferenciar por el verbo HTTP utilizado, los distintos endpoints e incluso el tipo y valor de la respuesta.

Peor Caso Black-Box

En este caso tomamos la API *session-service* para comparar los distintos nombres de los casos de test generados. Veamos por ejemplo los siguientes casos de la suite de tests *EvoMaster_faults_Test.java*:

- test_0_deleteOnSessionShowsFaults_100_200
- test_1_getOnSessionShowsFaults_100_200
- test_2_getOnSessionShowsFaults_100_200
- test_3_putOnSessionShowsFaults_100_200
- test_4_postOnSessionShowsFaults_100_200
- test_5_getOnQueryWithQueryParamsShowsFaults_100_200
- test_6_deleteOnSessionShowsFaults_100_200
- test_7_getOnSessionShowsFaults_100_200
- test_8_putOnSessionShowsFaults_100_200
- test_9_getOnQueryWithQueryParamsShowsFaults_100_200
- test_10_getOnSessionShowsFaults_100_200
- test_11_getOnQueryWithQueryParamsShowsFaults_100_200
- test_12_getOnQueryWithQueryParamsShowsFaults_100_200
- test_13_getOnQueryWithQueryParamsShowsFaults_100_200
- test_14_getOnSessionShowsFaults_100_200
- test_15_postOnFetchReturnsSchemaInvalidResponse
- test_16_postOnSessionReturnsSchemaInvalidResponse
- test_17_postOnFetchReturnsSchemaInvalidResponse
- test_18_postOnSessionReturnsSchemaInvalidResponse
- test_19_postOnFetchReturnsSchemaInvalidResponse
- test_20_postOnFetchReturnsSchemaInvalidResponse

Si bien en este caso hay muchos nombres repetidos, se pueden distinguir utilizaciones de recursos mencionados anteriormente, tales como las categorías de falta para los distintos errores. En un esfuerzo por desambiguar, no obstante, se pueden distinguir el verbo HTTP y la presencia de *query params*.

5. CONCLUSIÓN

Tal como fue presentado en el trabajo, se pudo incorporar Python como opción de salida sin que ello causara grandes dificultades. Lo cual permite pensar en continuar agregando distintos lenguajes de programación a EvoMaster. A su vez, al comparar con los resultados de la segunda parte del trabajo, se puede ver que el modo de ejecución *white-box* provee una suite de test más grande. Un posible eje de trabajo futuro puede ser entonces la incorporación de Python como opción *white-box*.

Por otro lado, se puede ver de forma inmediata como la utilización de las herramientas provistas por EvoMaster con su algoritmo genético generan nombres de test descriptivos. De forma tal que guíe a los usuarios a entender que tests fueron generados sin que haya que inspeccionar cada uno.

Un buen trabajo futuro de esta última sección es continuar investigando qué otras formas de desambiguar casos de test pueden utilizarse. Durante el desarrollo de este trabajo se realizó una prueba de concepto en la que los valores de los parámetros formaban parte de los nombres. Esto generaba nombres únicos, pero dado que EvoMaster realiza fuzzing, dichos parámetros a veces contenían valores que no aportan al nombre del test. Se puede continuar investigando así que formas de mejorar la experiencia de los usuarios para la generación automática de casos de test existen.

Es importante continuar la inversión en estas áreas, dado que la generación de casos de test es fundamental para la escalabilidad y correctitud del software desarrollado. Herramientas como EvoMaster proveen la capacidad de complementar el universo de búsqueda que el usuario puede tomar para así tender a un programa completo. Sin embargo, si bien crece el uso de estas herramientas, las mismas deben ser amigables para el uso por seres humanos dando opciones de lenguajes y tests declarativos que permitan seguir escalando y extendiéndose.

BIBLIOGRAFíA

[1] Andrea Arcuri. *EvoMaster: Evolutionary Multi-context Automated System Test Generation*. 2018.

[2] Andrea Arcuri. *Many Independent Objective (MIO) Algorithm for Test Suite Generation*. 2019.

[3] Bryan So Barton P. Miller Lars Fredriksen. *An Empirical Study of the Reliability of UNIX Utilities*. 1990.

[4] Man Zhang Bogdan Marculescu and Andrea Arcuri. *On the Faults Found in REST APIs by Automated Test Generation*. 2022.

[5] José Miguel Rojas Ermira Daka and Gordon Fraser. *Generating Unit Tests with Descriptive Names. Or: Would You Name Your Children Thing1 and Thing2*. 2019.

[6] EvoMaster. URL: `https://github.com/WebFuzzing/EMB`.

[7] OpenAPI Initiative. URL: `https://www.openapis.org/`.

[8] JetBrains. *Which programming, scripting, and markup languages have you used in the last 12 months?* URL: `https://www.jetbrains.com/lp/devecosystem-2023/languages/`.

[9] JSON. URL: `https://www.json.org/json-en.html`.

[10] Python. URL: `https://www.python.org/`.

[11] Python Requests. URL: `https://requests.readthedocs.io/en/latest/`.