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

Testing Automatizado de APIs REST basadas en Python Flask

Tesis de Licenciatura en Ciencias de la Computación

Axel Ezequiel Maddonni

Director: Juan Pablo Galeotti

Buenos Aires, 2022

TESTING AUTOMATIZADO DE APIS REST BASADAS EN PYTHON FLASK

En este trabajo se introduce una nueva extensión de EvoMaster, una herramienta open-source para generación automática de testeos impulsada por un algoritmo evolutivo, para aplicaciones web REST Python implementadas bajo el framework Flask. Esta herramienta cuenta con dos componentes principales: un *core*, encargado de llevar a cabo el algoritmo de búsqueda denominado MIO que aplica heurísticas para la optimización de testeos buscando maximizar la cobertura de líneas y branches sobre el código de la aplicación a testear; y un *driver* o controlador, encargado de alimentar al core con las métricas necesarias para la evolución del algoritmo. Se describe el diseño e implementación de un controlador exclusivo para Python encargado de la comunicación con el core bajo un protocolo ya establecido y la instrumentación de código mediante la aplicación de transformaciones sobre el árbol AST del código fuente original. Además, como parte de este trabajo se extiende el core para soportar la escritura de los tests en lenguaje Python y se provee de una librería Python para la distribución y testeo del nuevo controlador. Por último, se reportan y analizan métricas de cobertura de líneas y cobertura de branches obtenidas a partir de tests generados automáticamente para cinco casos de prueba diferentes, tres de ellos artificiales y dos de ellos originales de terceros, disponibles públicamente en GitHub.

Palabras claves: REST, Testing, APIs Web, Algoritmos Genéticos, Metaheurísticas, Python

AGRADECIMIENTOS

En primer lugar, quiero agradecer a JP, mi director, por su compromiso para ayudarme en este largo camino que fue la realización de esta tesis. Y por su paciencia en los momentos en los que no pude dedicarle el tiempo que ameritaba por cuestiones personales y laborales.

A los docentes de la facultad de los cuales aprendí muchísimo, principalmente a aquellos de gran calidad humana que lograban que aquellas clases teóricas tan pesadas se vuelvan más amenas; a los ayudantes que se quedaban horas respondiendo mis consultas, a mis compañeros de cursada que fui encontrando en el camino, porque en todo momento tuve la suerte de estar bien acompañado y de haber formado muy buenos grupos de estudio, que me hicieron más fácil transitar la cursada, y me ayudaron a disfrutarla como la disfruté a pesar de las corridas para llegar a las clases desde el trabajos. En especial a Lu y Manu, con las que compartí más todo el último tramo de la carrera, siempre apoyándonos mutuamente.

A la UBA y a la Facultad de Ciencias Exactas y Naturales, por permitirme el acceso a educación libre y gratuita pero de una calidad digna de ser destacada y distinguida entre las mejores. A la gente del departamento de computación y centro de estudiantes, por acompañarnos en los momentos donde no sabemos a quien acudir.

A mis amigos de toda la vida, que estuvieron presentes en las buenas y en las malas, y que me levantaban el ánimo cuando parecía que no podría alcanzar mis objetivos. A mi primo Fede, por haberme aconsejado elegir esta hermosa carrera, y apoyarme en todos estos años.

A mi familia, Andrea, Daniel y Giuli, que todos estos años vivenciaron mis logros y derrotas, pero que siempre me impulsaron a seguir adelante con pasión y dar lo mejor de mi para llegar a la meta.

Índice general

1. INTRODUCCIÓN

Las APIs REST [6] se han vuelto muy populares en la industria, particularmente en aplicaciones web desarrolladas en base a arquitecturas de microservicios. Un servicio web REST expone una API utilizando el protocolo HTTP, que permite realizar operaciones sobre recursos interactuando con bases de datos u otros servicios.

EvoMaster [2] es una herramienta diseñada para generar automáticamente tests de nivel de sistema para APIs REST. La herramienta implementa técnicas que aprovechan los lineamientos que siguen este tipo de aplicaciones para producir tests más adecuados. Estos tests tienen la forma de secuencias de requests/responses HTTP sobre el servicio en cuestión.

El "core" de EvoMaster está basado en un enfoque "white-box", es decir, se asume que se tiene acceso al código fuente de la aplicación que se quiere testear. La herramienta utiliza un algoritmo evolutivo para generar tests que son evaluados en base a información de coverage calculada en tiempo real, y a métricas sobre los distintos flujos (o "branches") de control de programa.

Actualmente EvoMaster cuenta con clientes (o "drivers") diseñados para realizar la instrumentación necesaria en el código de la aplicación, que permiten calcular las métricas que son consumidas por el "core" para la generación de los tests. Estos "drivers" o controladores están implementados para aplicaciones escritas en Java [10] y en Javascript [11], y permiten escribir los tests en el lenguaje correspondiente.

En esta tesis proponemos implementar un nuevo "driver" de EvoMaster para aplicaciones REST escritas en Python [16], implementadas usando el framework "Flask" [7] para desarrollo web. El nuevo "driver" deberá implementar la interfaz necesaria para comunicarse con el "core", instrumentar automáticamente código Python de aplicaciones Flask y exponer la información necesaria para generar y evolucionar los tests. Además, se deberá extender el "core" para poder traducir los nuevos tests a lenguaje Python, con el objetivo de facilitar la integración de los mismos con el código de la aplicación.

Como parte de la tesis, una vez finalizado el prototipo, realizaremos una evaluación empírica de los beneficios del enfoque white-box versus black-box para aplicaciones open source e industriales Flask siguiendo los lineamientos indicados en [3].

2. MARCO TEÓRICO

2.1. RESTful APIs basadas en Python Flask

La herramienta implementada en esta tesis fue diseñada para ser utilizada sobre aplicaciones que expongan APIs que sigan los lineamientos REST, y a su vez, hayan sido escritas en lenguaje Python sobre el framework de desarrollo web Flask.

REST o "Transferencia de Estado Representacional" hace referencia a un conjunto de principios de arquitectura de software, diseñados para construir servicios web sobre el protocolo HTTP. El concepto se origina a partir de una tesis doctoral en el año 2000 [6] y es ampliamente utilizado por las grandes desarrolladoras de software del mercado. Para que una aplicación sea considerada REST, debe cumplir una serie de requisitos:

- La aplicación se comunica con otros componentes usando una arquitectura cliente-servidor *stateless* o sin estado. Es decir, toda interacción con la aplicación contiene el contexto necesario para ser interpretada y procesada.
- La aplicación expone un conjunto de recursos identificados unívocamente por una URI ("Identificador uniforme de recurso"). Los recursos referencian conceptos o entidades, y pueden ser estáticos o dinámicos.
- Los recursos son representados por alguno de los formatos estándar como JSON, XML o HTML. Las representaciones incluyen datos de la entidad, meta-datos que describen los datos, y, en ocasiones, meta-datos para describir los meta-datos. Los requests o mensajes intercambiados también pueden incluir datos de control, que suelen ser utilizados para parametrizar mensajes, o modificar el comportamiento esperado al procesar el mensaje.
- Los recursos pueden ser manipulados a través de un conjunto de operaciones predefinidas. Las APIs web usan operaciones del protocolo HTTP para operar sobre un recurso. Las principales son:
  - `GET`, para obtener la representación del estado de un recurso.
  - `POST`, para procesar una representación de un recurso incluida en el request con una semántica específica.
  - `PUT`, para crear o modificar el estado de un recurso.
  - `DELETE`, para borrar el estado de un recurso.

Observemos el siguiente ejemplo de operaciones REST para un servicio que permite acceder a un catálogo de películas:

- `GET /peliculas` (devuelve el listado de películas en el catálogo)
- `GET /peliculas?genero=comedia` (devuelve el listado de películas filtradas por género)
- `POST /peliculas` (agregar una nueva película al catálogo)
- `GET /peliculas/{id}` (devuelve la película con el id provisto)

- `GET /peliculas/{id}/valoracion` (devuelve la valoración de una película específica)
- `DELETE /peliculas/{id}` (elimina una película específica del catálogo)

Existen muchos frameworks que facilitan la creación de APIs REST en cualquier lenguaje. En el caso del lenguaje Python, uno de los frameworks disponibles es Flask. A diferencia de otros frameworks más complejos como Django, Flask es denominado un "micro-framework" debido a su diseño minimal, y aunque no cuenta con tantas capacidades built-in como otros frameworks, es sencillo y rápido de usar, lo cual lo hace una alternativa atractiva para desarrollar microservicios en Python.

2.2. Search based testing

La generación de casos de prueba para testing es una tarea compleja, dado que el código de una aplicación puede ser arbitrariamente complejo. En el estado del arte, existen muchas estrategias para automatizar la generación de tests.

Una de las alternativas más simples se trata de generar tests aleatoriamente, la cual en general, es capaz de cubrir sólo una pequeña parte del sistema bajo testeo.

El problema de testing puede ser modelado como un problema de optimización, donde se busca maximizar el coverage del código de la aplicación y la cantidad de fallas alcanzadas por los testeos automatizados. Los algoritmos basados en búsqueda o "search-based algorithms" aplican metaheurísticas para intentar resolver este tipo de problemas.

En el caso de EvoMaster, se utiliza un algoritmo evolutivo. Este tipo de algoritmos evolutivos se basan en generar una población de individuos (posibles soluciones al problema) que se seleccionan en base a su valor de *fitness* y se reproducen mediante mutaciones u operaciones de *crossover* (cruza entre individuos) evolucionando hasta una solución óptima o hasta que el tiempo dedicado se agote.

En el algoritmo introducido por EvoMaster [2], los individuos de la población serán los tests a generar, y se evolucionan intentando cubrir una serie de objetivos o *targets* (como cubrimiento de líneas o ramas de código).

2.3. EvoMaster

EvoMaster es una herramienta *open-source* que permite generar automáticamente testeos de sistema para aplicaciones REST en Java y JavaScript. Internamente, la herramienta ejecuta un algoritmo evolutivo y análisis dinámico de programas, para evolucionar y generar testeos efectivos que intentan maximizar el coverage y la detección de fallas.

Para poder utilizar esta herramienta, la API bajo test deberá exponer su documentación usando OpenAPI [18]. A partir de ésta, se obtiene la información necesaria sobre los endpoints disponibles que deberán ser testeados, como así también las distintas operaciones que pueden realizarse con cada uno de ellos, con sus respectivos parámetros y tipos. Los individuos del algoritmo evolutivo se definen en base a templates generados a partir de esta documentación.

Existen dos modos de ejecución de la herramienta: *white-box testing* y *black-box testing*. El modo *black-box* permite generar tests para cualquier API, sea cual sea el lenguaje, sin necesidad de tener acceso al código fuente, siempre y cuando exponga su esquema. En el modo *white-box*, en cambio, se aprovecha el acceso al código fuente de la aplicación a

testear para obtener métricas en runtime a partir de una versión instrumentada de código, que será utilizada por el algoritmo evolutivo para evolucionar las soluciones parciales. Las ejecuciones *white-box* generalmente dan mejores resultados que las *black-box*, dado que no permiten realizar ningún análisis sobre el código.

La arquitectura de EvoMaster se divide principalmente en un componente core y un componente driver o controlador. El core es el encargado de llevar a cabo el algoritmo evolutivo y es agnóstico del tipo de lenguaje de la aplicación a testear, a excepción de la escritura de los tests, que deberían escribirse en el mismo lenguaje de la aplicación bajo testeo, aunque nada impide que sean escritos en uno diferente. En cambio, el controlador, tiene acceso al código fuente (sólo es utilizado en testeos *white-box*) y hará de nexo entre el core y la aplicación. El controlador se encarga de instrumentar el código fuente, identificar los objetivos a cubrir, y recolectar y enviar las métricas necesarias al core para la evolución de los tests. Además, el core maneja el estado de la aplicación a través del controlador.

Los testeos generados por EvoMaster son auto-contenidos y pueden ser usados como testeos de regresión. Los mismos llevan una estructura similar: generación de datos de entrada, ejecución de *requests* y por último, aserciones o chequeos sobre los resultados. Además, se intenta maximizar la detección de fallas o *bugs* en el sistema usando distintas heurísticas, por ejemplo, identificando *requests* que devuelvan un *status* 500 ("Internal Server Error").

Existen otros *features* disponibles en la herramienta que no serán alcanzados por la implementación de este trabajo, como el soporte para bases de datos. La herramienta permite generar datos automáticamente en la base de datos previo a la ejecución de los tests. Dicha información se calcula interceptando las comunicaciones de la aplicación con la misma.

2.3.1. Introducción al algoritmo de EvoMaster: MIO

Para presentar el algoritmo evolutivo usado por EvoMaster denominado MIO (*Many Independent Objective*), debemos definir cuáles son los objetivos a cubrir del mismo. Los objetivos referencian elementos del código a cubrir con los testeos generados:

- líneas de código
- branches de código (definidos por estructuras de flujo)
- status codes devueltos por los distintos endpoints

El algoritmo intentará maximizar la cobertura de estos objetivos, asumiendo que: 1) Siempre podremos agregar nuevos testeos para aumentar la cobertura 2) Algunos objetivos pueden estar relacionados, por ejemplo, en múltiples estructuras de control encadenadas, como condicionales o ciclos. 3) Algunos objetivos pueden ser imposibles de cubrir.

En el algoritmo, se evoluciona cada población de individuos o tests para cada uno de los objetivos a cubrir, en base a su valor de *fitness* calculado exclusivamente para cada objetivo.

Al inicio del algoritmo cada población se inicializa vacía, y en cada iteración, con cierta probabilidad se elige entre generar nuevos testeos aleatoriamente, o realizar un sampleo y mutación de alguno de los testeos de los objetivos pendientes a cubrir. El nuevo testeo generado por dicha mutación se agrega a cada una de las poblaciones de objetivos pendientes, y se evalúan independientemente en cada una de ellas.

Para conocer los objetivos alcanzados por un nuevo individuo, el algoritmo puede hacer uso de las respuestas obtenidas por los requests enviados al ejecutar un testeo y medir por ejemplo, cuáles son los status codes obtenidos. Y en caso de tener acceso al código fuente, hacer uso de métricas de cobertura para saber cuáles fueron los statements o branches alcanzadas. Dichas métricas se obtienen a partir de la instrumentación del código, que se explicará con detalle en la siguiente sección.

Cuando se alcanza un tamaño máximo de población para algún objetivo, se elimina el peor test, nuevamente en base a su `fitness`. Cuando se cubre alguno de los objetivos, se actualiza el resultado con el testeo seleccionado para dicho objetivo y su población deja de usarse para el sampleo, descartando el resto de los individuos almacenados en la misma. Al finalizar la búsqueda, se genera una suite de testeos en base a los mejores testeos generados en cada población, eliminando los repetidos en caso de que existan.

El MIO mantiene un contador por cada población de tests, que se incrementa cada vez que un test se genera a partir del sampleo de dicha población y se resetea cada vez que se agrega un mejor test a la misma. En las iteraciones donde se samplean tests para generar un nuevo individuo mutado, se elige la población cuyo contador contiene el menor valor. Esta técnica se denomina "sampleo dirigido por el feedback" y permite evitar evolucionar las poblaciones de objetivos que no se pueden cumplir.

Además, en un momento del algoritmo, el MIO comienza lo que se denomina "búsqueda enfocada" o *focus search*, donde se dejan de crear tests aleatorios y se enfoca sólo en evolucionar las poblaciones actuales usando mutaciones. Esto es similar a lo realizado por la meta-heurística denominada "Simulated Annealing", donde se intenta hacer un balance entre la exploración del espacio de búsqueda para luego centrarse en la explotación de un espacio particular, que es útil cuando el espacio de búsqueda del problema es muy grande. Esta técnica se lleva a cabo mediante la actualización de los parámetros de búsqueda en cada iteración. Por ejemplo, en el momento en que comienza la búsqueda enfocada, el tamaño máximo de población se reduce a 1, y la probabilidad de generar individuos aleatorios baja a 0.

A continuación se presenta un pseudocódigo del algoritmo:

Algorithm 1 Pseudocódigo del algoritmo de búsqueda MIO

Require: Condición de Parada: C, Función de fitness: δ, Tamaño máximo de población: n, Probabilidad de sampleo aleatorio: $P_{sampleo}$, Comienzo de "focus search": F

```
P ← inicializarPoblaciónVacía()
R ← {}
while ¬C do
    if Prob_sampleo > rand() then
        i ← generarIndividuoAleatorio()
    else
        i ← samplearIndividuo(P)
        i ← mutarIndividuo(i)
    end if
    for o ∈ objetivosAlcanzados(i) do
        if objetivoCubierto(o) then
            actualizarResultado(R, i)
            P ← P − P_o
        else
            P_o ← P_o ∪ {i}
            if |P_o| > n then
                quitarPeorIndividuo(P_o, δ)
            end if
        end if
    end for
    actualizarParámetros(F, Prob_sampleo, n)
end while
return R
```

2.3.2. Representación del problema

Para entender cómo se aplica el algoritmo MIO sobre el dominio de problema de testing, debemos entender cómo se representan las posibles soluciones al problema, operadores sobre las mismas y la función de *fitness* utilizada durante la búsqueda.

En el contexto de testing sobre APIs REST, una posible solución consta de una o más llamadas HTTP hacia el servicio bajo testeo. Dicha llamada se representa con los siguientes elementos:

- Operador HTTP
- Headers HTTP
- Path o endpoint
- Parámetros (enviados en la URL)
- Payload (enviados en el body del request), que puede expresarse en diversos formatos, como JSON o XML.

Para conocer los distintos *path* disponibles en la API, EvoMaster hace uso de la especificación OpenAPI[18] que el servicio expone. Dicha especificación, anteriormente conocida

como Swagger, se ha convertido en un estándar para documentar aplicaciones REST, independientemente del lenguaje utilizado para desarrollarlas. Este tipo de documentación facilita el descubrimiento y entendimiento de las capacidades de un servicio sin necesidad de acceder al código fuente o a documentación adicional.

Una vez parseada la especificación de la API, para cada endpoint se crea una serie de templates de cromosomas, que contienen información inmutable (como el path que representa un recurso) e información mutable denominados genes (como los query parameters, los headers o el body payload del request). Existen distintos tipos de genes como: *Boolean*, *Integer*, *String*, etc. y pueden definir restricciones sobre sus posibles valores (como por ejemplo, un rango de valores numérico).

Al samplear un nuevo test case, se genera un número aleatorio de llamadas HTTP en base a dichos templates, con valores aleatorios de genes en base a las restricciones definidas. Las mutaciones realizadas por el algoritmo pueden modificar la estructura del test, o los valores de los genes de alguna de sus llamadas HTTP.

2.3.3. Extensiones de EvoMaster

Al momento de comenzar este trabajo, EvoMaster contaba con extensiones de controladores para los lenguajes Java y JavaScript. Estas librerías proveen funcionalidades para instrumentar el código de las aplicaciones bajo testeo, y una API para exportar la información de cobertura en formato JSON. Esta información es luego consumida por la herramienta que implementa el algoritmo MIO para calcular el *fitness* sobre los tests generados por el mismo en cada iteración. De esta manera, el core que implementa el algoritmo es independiente de la implementación de los controladores, que dependen del lenguaje de programación del SUT para instrumentar el mismo.

Este enfoque donde el algoritmo se ejecuta separado de la herramienta encargada de la instrumentación puede ser costoso, dado que se agrega el overhead de las llamadas HTTP para obtener la información de cobertura durante la ejecución del MIO. Sin embargo, es aceptable para la generación de system tests ya que, a diferencia de unit tests, los mismos usualmente interactúan con servicios externos y el overhead no impacta tanto en la performance. Igualmente, para evitar reducir dicho costo, los controladores de EvoMaster permiten reducir el scope del código a cubrir mediante un parámetro que define los paquetes o módulos a instrumentar. Además, al obtener la información de coverage sólo se envían las métricas correspondientes a los nuevos objetivos cubiertos, o de aquellos para los cuales se obtuvo una mejor *branch distance*.

El primer controlador de Evomaster fue diseñado con un foco en lenguajes basados en JVM, como Java. Para realizar la instrumentación del código se utilizaron *Java Agents* que durante la carga de las clases del sistema bajo testeo se encargan de insertar código de instrumentación que permite observar y recolectar información en tiempo de ejecución, que luego será utilizada para calcular y reportar las métricas de coverage al core de Evomaster. El controlador Java, además de instrumentar el código, provee una funcionalidad para el inicio y apagado del SUT en un puerto TCP, que es utilizada por los tests al momento de ser ejecutados. Los tests realizan las aserciones usando la librería *RestAssured*, aplicando chequeos sobre los códigos HTTP obtenidos por cada request.

Por otro lado, la extensión del controlador para JavaScript también realiza una instrumentación del código, aplicando heurísticas similares pero haciendo uso de la herramienta *Babel*, que permite transformar código JavaScript. Babel es un traspilador mayormente utilizado para convertir código JavaScript que usa sintaxis más modernas (como ES6+),

no completamente soportadas por navegadores, a versiones retro-compatibles de JavaScript. Para llevar a cabo esto, esta herramienta primero se encarga de parsear el código fuente, obtener la representación AST del mismo, transformarlo, y luego generar el nuevo código a partir del árbol modificado.

En la siguiente sección se detalla la implementación realizada para extender EvoMaster para el lenguaje Python. La estrategia para llevar a cabo la instrumentación de código Python se asemejará a la utilizada por el controlador para JavaScript, aplicando transformaciones sobre el árbol AST del código fuente original.

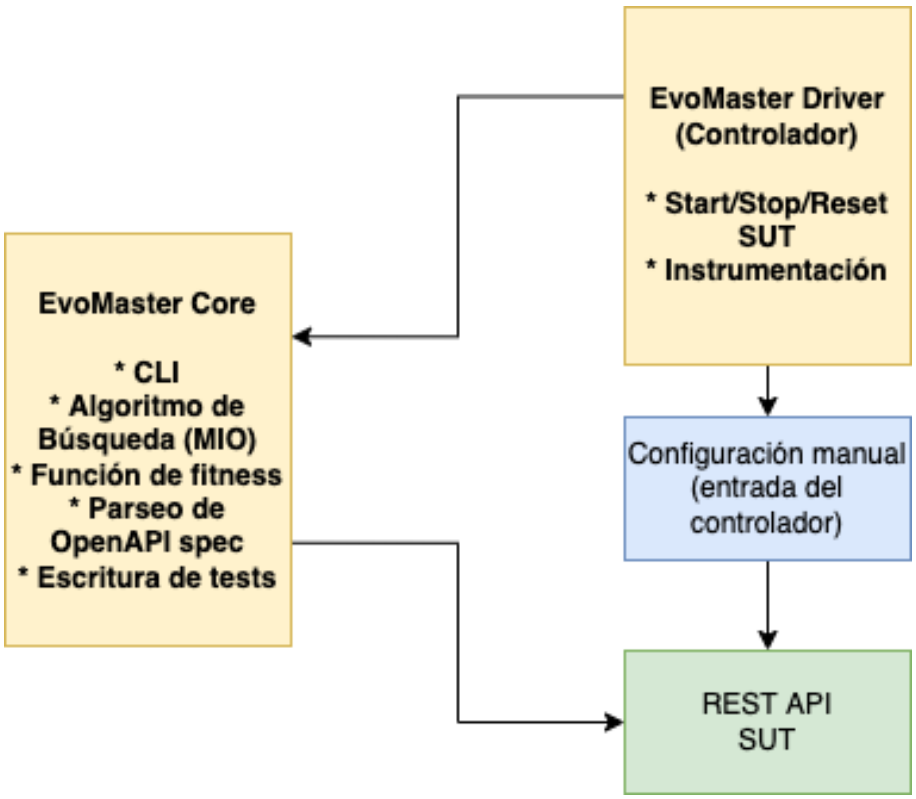

Fig. 2.1: Arquitectura de EvoMaster

3. IMPLEMENTACIÓN

En esta sección se presenta una solución que implementa una extensión de EvoMaster para la automatización de la generación de tests para aplicaciones REST Python desarrolladas usando el framework Flask.

3.1. Instrumentación del código

La función de *fitness* utilizada por EvoMaster en el algoritmo evolutivo de búsqueda se define en base a tres objetivos: (1) *statement coverage*, (2) *branch coverage* y (3) *HTTP status codes* devueltos por las APIs del sistema bajo test. Cada uno de estos objetivos es representado con un valor entre 0 y 1, que define cuán cerca a ser completado se encuentra. Cuando dos casos de test computan la misma cobertura, el algoritmo MIO selecciona el caso más corto.

A diferencia de los *status codes*, que son simplemente obtenidos de cada respuesta que llega al core de EvoMaster durante su ejecución, las métricas de *coverage* deben ser computadas por el nuevo controlador de EvoMaster para Python. Este controlador se encargará de instrumentar el sistema bajo testeo en tiempo real, antes de comenzar la generación de los tests. La instrumentación se basa en agregar código inmediatamente antes y/o después de cada condición de *branching* dentro del código de la aplicación, para observar los valores obtenidos durante la ejecución de los tests. Estos valores se exponen al core a través de una API para medir el *fitness* de cada individuo generado por el algoritmo evolutivo.

Sin embargo, medir el *coverage* no es suficiente debido a partes del código que pueden encontrarse en flujos de control alcanzados cuando se cumplen ciertos predicados, que pueden ser difíciles de cubrir con parámetros generados aleatoriamente. Para ello se utiliza una heurística denominada *branch distance* que mide cuán lejos se encuentran de cumplir una condición para ingresar a una rama del código particular. Para calcularla, la instrumentación del código consiste en reemplazar métodos booleanos por otros que calcularán el valor de la *branch distance* en base a ciertas heurísticas que definiremos a continuación.

Para instrumentar aplicaciones en Python, se utilizó una estrategia que consiste en recorrer el AST ("Árbol de sintaxis abstracta") del código a testear. Estas estructuras consisten en representaciones del código Python parseado en forma de árboles, y suelen ser usados para análisis semántico y generación de código. Los ASTs contienen información sobre la estructura y semántica del código, además de los *tokens* utilizados en el código fuente original (como el nombre de las variables y las funciones). Usando esta estrategia, el controlador modificará el árbol AST de la aplicación en *runtime*, para agregar llamadas a funciones encargadas de calcular y almacenar las métricas de *coverage*. Python, a partir de su versión 2.6, provee un módulo `ast` que permite modificar dichos árboles.

El módulo `ast` provee una clase `NodeTransformer`, que implementa el patrón *Visitor*, y puede ser utilizada para recorrer el AST de un módulo Python llamando a una función definida para cada tipo de nodo dentro del árbol, permitiendo modificaciones sobre los mismos. Los tipos de nodos que nos interesan para la instrumentación son los siguientes: Nodos tipo `Statement`, que incluyen asignaciones, aserciones, definiciones de funciones y clases, operadores para control de flujo, manejo de excepciones, `import`s, `return`s, entre

otros. Nodos tipo `Module`, que representan módulos de Python. Nodos tipo `Compare`, que representan comparaciones entre dos o más valores usando una lista arbitraria de comparadores que incluyen: `Eq`, `NotEq`, `Lt`, `LtE`, `Gt`, `GtE`, `Is`, `IsNot`, `In`, `NotIn`. Nodos tipo `UnaryOp`, que representan operaciones unarias, entre las cuales, sólo cubriremos la operación `Not`. Y, por último, nodos tipo `BoolOp`, que representan a las operaciones booleanas `And` y `Or`. Se define una función visitante para cada una de estas clases de nodos:

En primer lugar, el visitor de `Module`, se encargará de agregar los `import`s de las funciones necesarias para la instrumentación, que se utilizan en el resto de los visitors.

En segundo lugar, el visitor de `Statement` es el encargado de la instrumentación del coverage. Por cada statement en el código de la aplicación, se agrega una función antes y después de los mismos: "`entering_statement`" y "`completed_statement`", encargadas de marcar dichos nodos como visitados en una estructura global, que servirá de almacenamiento de métricas de coverage para una determinada ejecución. Para nodos que representan `return` statements, dado que no hay manera de llegar a ejecutar una función luego del `return`, se utiliza una única función "`completion_statement`", que además de marcarlo como visitado, retornará el valor correspondiente. Para el caso de controles de flujo (como `If`, `For`, `While`), dado que en los mismos pueden encontrarse `return`s, tampoco se utilizan "`completed_statement`", y se considerarán como completados con sólo acceder a los mismos.

El visitor de `Compare` reemplazará cada tipo de comparación por otra función que, además de ejecutar la comparación original y retornar su resultado, calculará un valor de *branch distance* para dicha comparación usando una heurística definida en base al tipo de datos siendo comparados. Para cada comparación, se estiman dos valores entre 0 y 1, que representan las distancias de cada una a resultar en `True` y a `False` respectivamente. A la combinación de estos dos valores denominaremos "`Truthness`" de una expresión. Notar que uno de estos valores siempre será 1, ya que cada comparación resultará en alguno de estos dos valores lógicos.

A continuación, se muestra cómo calcular cada valor de `Truthness` (la *branch distance* a cada valor booleano), en base al valor de cada operando y al tipo de operación. Notar que para algunos operadores, reusaremos fórmulas de `Truthness` previamente definidas para otros operadores.

Las heurísticas implementadas para calcular el `Truthness` de comparaciones aplican solamente a operandos de tipo numérico, que en Python incluyen los tipos: `int`, `float` y `complex` - abreviaremos `number` para referirnos a ellos - o cadenas de caracteres, representados en Python por el tipo `str`.

Para dos variables del mismo tipo:

$$x_1, x_2 \in T$$

$$T \in \{number, str\}$$

definimos las siguientes heurísticas:

$$Truthness_{ofTrue}(x_1, x_2, Eq) = \begin{cases} 1 & x_1 = x_2 \\ 1 - norma(distancia_T(x_1, x_2)) & x_1 \neq x_2 \end{cases}$$

$$Truthness_{ofFalse}(x_1, x_2, Eq) = \begin{cases} 0 & x_1 = x_2 \\ 1 & x_1 \neq x_2 \end{cases}$$

$$Truthness_{ofTrue}(x_1, x_2, Lt) = \begin{cases} 1 & x_1 < x_2 \\ 1 - norma(distancia_T(x_1, x_2)) & x_1 \geq x_2 \end{cases}$$

$$Truthness_{ofFalse}(x_1, x_2, Lt) = \begin{cases} 1 - norma(distancia_T(x_1, x_2)) & x_1 < x_2 \\ 1 & x_1 \geq x_2 \end{cases}$$

Definimos la función `distancia` para cada tipo como:

$$distancia_{number}(n_1, n_2) = |n_1 - n_2| \qquad n_1, n_2 \in number$$

$$distancia_{str}(s_1, s_2) = |s_1.length - s_2.length| * C + \sum_{i=0}^{min(s_1.length, s_2.length)} |s_{1_i}.ord - s_{2_i}.ord| \qquad s_1, s_2 \in str$$

Y la función `norma` de la siguiente manera[1]:

$$norma(x) = \frac{x}{x+1} \qquad x \in R$$

Definimos la función auxiliar `invertir` que nos permite definir el `Truthness` para el resto de las operaciones:

$$invertir(Truthness_{ofTrue}(x_1, x_2, op)) = Truthness_{ofFalse}(x_1, x_2, op)$$

$$invertir(Truthness_{ofFalse}(x_1, x_2, op)) = Truthness_{ofTrue}(x_1, x_2, op)$$

$$Truthness(x_1, x_2, NotEq) = invertir(Truthness(x_1, x_2, Eq))$$

$$Truthness(x_1, x_2, GtE) = invertir(Truthness(x_1, x_2, Lt))$$

$$Truthness(x_1, x_2, LtE) = invertir(Truthness(x_2, x_1, Lt))$$

$$Truthness(x_1, x_2, Gt) = invertir(Truthness(x_1, x_2, LtE))$$

Para el visitor de `UnaryOp`, se aplicará una heurística que simplemente invertirá el valor de la `Truthness` de la expresión alcanzada por el operador. En caso de tratarse de una negación de una constante o una variable, no se computarán *branch distances* para dicha operación.

Por último, el visitor de `BoolOp` aplicará una heurística definida a partir de las `Truthness` de cada uno de los dos operandos:

Sean:

$$x, y \in bool$$

definimos las siguientes heurísticas:

$$Truthness_{ofTrue}(x, y, And) = \frac{T_{ofTrue}(x)}{2} + \frac{T_{ofTrue}(y)}{2}$$

$$Truthness_{ofFalse}(x, y, And) = max\{T_{ofFalse}(x), T_{ofFalse}(y)\}$$

$$Truthness_{ofTrue}(x, y, Or) = max\{T_{ofTrue}(x), T_{ofTrue}(y)\}$$

$$Truthness_{ofFalse}(x, y, Or) = \frac{T_{ofFalse}(x)}{2} + \frac{T_{ofFalse}(y)}{2}$$

Notar que todas las definiciones para las heurísticas toman sólo dos variables u operandos. En caso de tener predicados que involucren más de dos operandos, éstos no serán instrumentados. La extensión de dichas heurísticas a más de dos operandos es factible, aunque se decide dejarla por fuera del scope de este trabajo.

Es importante notar que, al comparar otros tipos de datos, o comparar cadenas de caracteres con valores numéricos por igualdad, la implementación actual no permite calcular un valor de branch distance y simplemente se retorna un truthness en base al resultado booleano de la operación, es decir:

$$Truthness_{ofTrue}(x, y, Op) = \begin{cases} 1 & Op(x, y) == True \\ 0 & Op(x, y) == False \end{cases}$$

$$Truthness_{ofFalse}(x, y, Op) = \begin{cases} 0 & Op(x, y) == True \\ 1 & Op(x, y) == False \end{cases}$$

A continuación, veamos un ejemplo de cálculo de `Truthness` para la siguiente expresión:

$$a > 0 \quad \text{or} \quad b == \text{‘yes’}$$

Calcularemos el `Truthness` para cada valor de verdad por separado, tomando como valores $a = 10$ y $b =$ ‘no’.

Primero, aplicamos las fórmulas de `Truthness` para `Or` de la siguiente manera:

$$\begin{aligned} T_{ofTrue}(a > 0, b == \text{‘yes’}, Or) =& max\{T_{ofTrue}(a > 0), T_{ofTrue}(b == \text{‘yes’})\} \\ =& max\{T_{ofTrue}(a, 0, Gt), T_{ofTrue}(b, \text{‘yes’}, Eq)\} \end{aligned}$$

$$\begin{aligned} T_{ofFalse}(a > 0, b == \text{‘yes’}, Or) =& \frac{T_{ofFalse}(a > 0)}{2} + \frac{T_{ofFalse}(b == \text{‘yes’})}{2} \\ =& \frac{T_{ofFalse}(a, 0, Gt)}{2} + \frac{T_{ofFalse}(b, \text{‘yes’}, Eq)}{2} \end{aligned}$$

Calculamos el `Truthness` de cada sub-expresión por separado, reemplazando las variables por los valores mencionados:

$$
\begin{aligned}
T_{ofTrue}(10,0,Gt) =& T_{ofFalse}(10,0,LtE)\\
=& T_{ofTrue}(0,10,Lt)\\
=& 1\\
T_{ofFalse}(10,0,Gt) =& T_{ofTrue}(10,0,LtE)\\
=& T_{ofFalse}(0,10,Lt)\\
=& 1 - norma(distancia(0,10))\\
=& 1 - \frac{10}{11}\\
=& \frac{1}{11}\\
T_{ofTrue}(\text{`no'},\text{`yes'},Eq) =& 1 - norma(distancia(\text{`no'},\text{`yes'}))\\
=& 1 - norma(|\text{`no'}.length - \text{`yes'}.length| * C\\
& + \sum_{i=0}^{min(\text{`no'}.length,\text{`yes'}.length)} |\text{`no'}_{1_i}.ord - \text{`yes'}i.ord|)\\
=& 1 - norma(1 * C + |ord(n) - ord(y)| + |ord(o) - ord(e)|)\\
=& 1 - norma(1 * C + |110 - 121| + |111 - 101)|)\\
=& 1 - norma(1 * C + 21)\\
=& 1 - \frac{C + 21}{C + 22}\\
=& \frac{1}{C + 22}\\
T_{ofFalse}(\text{`no'},\text{`yes'},Eq) =& 1
\end{aligned}
$$

Luego, reemplazando en las expresiones originales:

$$
\begin{aligned}
T_{ofTrue}(a > 0, b == \text{`yes'}, Or) =& max\{1, \frac{1}{C + 22})\} && = 1\\
T_{ofFalse}(a > 0, b == \text{`yes'}, Or) =& \frac{\frac{1}{11}}{2} + \frac{1}{2} && = \frac{6}{11}
\end{aligned}
$$

3.2. Import Hooks

Para instrumentar el código de aplicaciones Python, esta nueva extensión de EvoMaster hace uso de una funcionalidad provista por Python denominada *Import Hooks*. Dicha funcionalidad permitirá modificar el código dinámicamente a medida que es importado. Introducidos en `PEP 302`[12], los import hooks son llamados cada vez que un módulo que todavía no había sido cargado es importado; y permiten modificar los mecanismos estándar de importación.

La implementación del protocolo de `ImportHooks` consiste en dos objetos fundamentales: un `MetaPathFinder` - que permite definir una especificación particular para cargar un módulo (denominados `module spec`) - y un `FileLoader` que provee la implementación para la carga de un módulo.

En el caso de EvoMaster, un nuevo `Loader` (al que llamaremos `InstrumentationLoader`) es el encargado de modificar el AST del código original y cargar el nuevo módulo instrumentado, mientras que el nuevo `InstrumentationFinder` definirá para qué módulos aplicará la instrumentación.

3.3. Controlador de EvoMaster

En esta sección se describe la interfaz implementada por el controlador de EvoMaster para comunicarse con el core. El controlador expone una API REST por la cual se puede controlar el estado del SUT ("System Under Test" o "Sistema bajo test"), marcar el inicio y fin de cada búsqueda del algoritmo evolutivo, y obtener las métricas recolectadas por el código instrumentado.

3.3.1. Protocolo

A continuación se explica brevemente el protocolo de comunicación entre el core y un driver de EvoMaster3.3.1. La interfaz de los controladores de EvoMaster es genérica, sin importar el lenguaje de programación del sistema bajo testeo. Es importante notar que algunos endpoints pueden no estar soportados por una versión del controlador, pero eso no impide el uso efectivo de mismo.

Cuando se inicia una nueva ejecución de EvoMaster para la generación de nuevos tests, el core se comunica con el controlador para iniciar el sistema instrumentado usando el endpoint `/controller/api/runSut`. Luego, para denotar el inicio de una búsqueda, se llama a `/controller/api/newSearch`. Con esta acción, se reinicia el estado interno del controlador para dar inicio a una nueva búsqueda independiente.

El controlador lleva un registro de cada request realizado hacia el SUT. Para esto, el core le envía un request al endpoint `/controller/api/newAction` por cada acción tomada.

El endpoint `/controller/api/testResults` permite al core obtener la información necesaria para calcular el *fitness*, para cada uno de los objetivos registrados por el controlador. Los objetivos incluyen métricas de *coverage* para cada módulo, línea de código y *branch* que hayan sido instrumentados por el controlador.

Este intercambio de mensajes entre el core y el controlador se repite las veces que sea necesario por el tiempo que haya sido definido al iniciar la ejecución.

Una vez finalizada la ejecución del algoritmo evolutivo, el core continúa con la escritura de los tests.

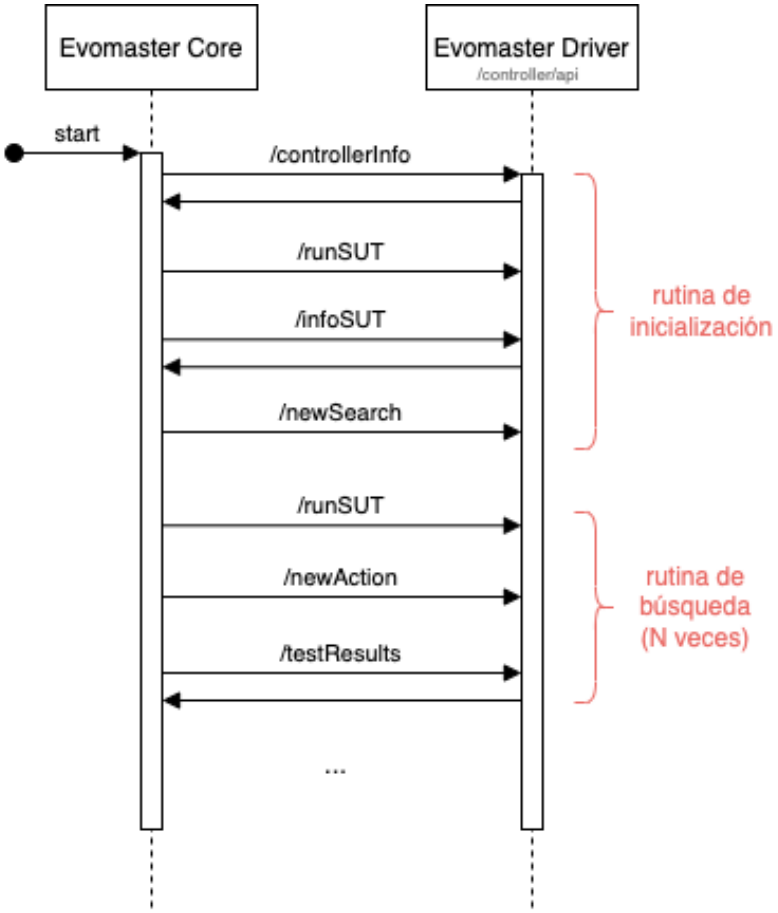

Fig. 3.1: Diagrama de secuencia del protocolo de comunicación de Evomaster

3.3.2. Escritura de tests

Previo a este trabajo, el core soportaba escritura de tests en código Java y Javascript. Se extendió el core para escribir tests en Python usando el módulo `unittest`.

Para la generación de datos de entrada, se usan tipos de datos nativos de Python: listas, diccionarios y tipos de datos primitivos; mientras que las aserciones se realizan usando el keyword `assert`.

La ejecución de los requests difiere levemente entre ejecuciones *black-box* y *white-box*. En el caso de las *white-box*, el test se encarga de levantar el sistema bajo testeo para efectuar los requests definidos por el test, limpiar la base datos en caso de ser necesario, y apagarla cuando finaliza el test. En cambio, para los tests *black-box*, los requests se ejecutan directamente usando la url donde se encuentra corriendo la aplicación previamente inicializada a mano. Todos los requests se realizan utilizando la librería `requests` para Python.

A continuatión se encuentran algunos ejemplos de tests generados automáticamente por Evomaster para Python:

```
1  import json
2  import requests
3  import unittest
4  from evomaster_client.em_test_utils import *
5
6  import evomaster_benchmark.em_handlers.ncs
7
8
9  # This file was automatically generated by EvoMaster on
   ↪  2022-04-02T14:26:33.848-03:00[America/Argentina/Buenos_Aires]
10 #
11 # The generated test suite contains 30 tests
12 #
13 # Covered targets: 383
14 #
15 # Used time: 0h 30m 0s
16 #
17 # Needed budget for current results: 17%
18 #
19 # This file contains test cases that represent successful calls.
20 class EvoMaster_successes_Test(unittest.TestCase):
21
22     controller = evomaster_benchmark.em_handlers.ncs.EMHandler()
23
24     @classmethod
25     def setUpClass(cls):
26         cls.controller.setup_for_generated_test()
27         cls.baseUrlOfSut = cls.controller.start_sut()
28
29     @classmethod
30     def tearDownClass(cls):
31         cls.controller.stop_sut()
32
33     def setUp(self):
```

```
34          self.controller.reset_state_of_sut()
35
36      def test_0(self):
37          headers = {}
38          headers['Accept'] = "*/*"
39          res_0 = requests \
40                  .get(self.baseUrlOfSut +
                    ↪  "/api/expint/1/0.17825056429089225",
41                      headers=headers)
42
43          assert res_0.status_code == 200
44          assert res_0.json() == json.loads("{\"resultAsFloat\":
            ↪  1.3179610584531232}\n")
45
46      # ...
```

Notar que los tests se encuentran contenidos dentro de una clase extendiendo de `unittest.TestCase`. Al inicializar la clase, se utiliza el controller para inicializar el sut llamando a `start_sut()` y al finalizar se termina llamando a `stop_sut()`. Entre cada test, se reinicia el estado del sut para comenzar cada testeo en el mismo estado inicial llamando a `reset_state_of_sut()`. Dicho método debe ser implementado por el usuario encargado de escribir el driver.

El test que se puede ver en el ejemplo es parte del grupo de tests generados para casos donde la respuesta esperada son respuestas exitosas (código http 200). Además de este grupo de tests, es posible generar automáticamente tests para otros tipos de respuesta como el que se puede ver a continuación, donde se envían parámetros no soportados por la API:

```
1       def test(self):
2
3           headers = {}
4           headers['Accept'] = "*/*"
5           res_0 = requests \
6                   .get(self.baseUrlOfSut + "/api/expint/-16135/0.0188",
7                       headers=headers)
8
9           assert res_0.status_code == 404
```

El driver para Python también soporta escritura de tests para aplicaciones con estado. En ese caso, múltiples requests pueden ser generados por el algoritmo para testear múltiples operaciones sobre un mismo objeto del dominio. Por ejemplo, el siguiente caso corresponde a un test que crea un objeto en una aplicación stateful y luego lo borra usando los métodos http POST y DELETE respectivamente:

```
1       def test(self):
2           headers = {}
3           body = {}
4           headers['Content-Type'] = "application/json"
5           body = " { " + \
```

```
            " \"text\": \"zjTuJbB\", " + \
            " \"country\": \"RYqxmt0VFX\" " + \
            " } "
        headers['Accept'] = "application/json"
        res_0 = requests \
                .post(self.baseUrlOfSut + "/news",
                    headers=headers, data=body)
        location_news = "/news/" + str(res_0.json()['id'])

        assert res_0.status_code == 200
        assert res_0.json() == json.loads("{\"authorId\": null, \"text\":
        ↪  \"zjTuJbB\", \"country\": \"RYqxmt0VFX\", \"id\": 1}\n")

        headers = {}
        headers['Accept'] = "*/*"
        res_1 = requests \
                .delete(resolve_location(location_news, self.baseUrlOfSut +
                ↪  str("/news/70")),
                    headers=headers)

        assert res_1.status_code == 204
        assert not res_1.text
```

Correctitud de los tests

Es importante observar que uno de los desafíos en la generación automática de tests es definir si un test generado es correcto o no. La manera más confiable sería contar con un desarrollador que revise las aserciones realizadas en cada test, pero esto no escala. Una manera muy básica es considerar que el SUT no crashee durante la ejecución del test, pero no todos los bugs causan un crash. Para system tests, sin embargo, un status code 500 podría significar algún comportamiento inesperado en la aplicación que requiera revisión y suele resultar una buena heurística para identificar bugs.

Aún así, los testeos generados automáticamente siguen siendo útiles para realizar testeos de regresión. Este tipo de testeos permiten identificar cambios inesperados en el comportamiento de la aplicación entre releases. Un claro ejemplo donde esto es útil es en seguridad. Un cambio en el status code devuelto por un endpoint puede significar un cambio en los requisitos de acceso al mismo.

3.3.3. Packaging

La implementación de este nuevo cliente de EvoMaster para Python se realizó en Python3 y se encuentra empaquetada y publicada en el Índice de paquetes de Python (PyPI [13]), el repositorio oficial para paquetes Python desarrolados por terceros. La documentación oficial y las instrucciones de su descarga y funcionamiento se pueden encuentrar en `https://pypi.org/project/evomaster-client/`.

Una vez instalado, se puede hacer uso de la misma mediante un CLI que expone los siguientes comandos:

1. `run-instrumented`, que permite correr una API Flask en su versión instrumentada.
2. `run-em-handler`, que permite correr un handler de EvoMaster que actuará de intermediario entre el core (que se debe ejecutar aparte) y la API bajo testeo en su versión instrumentada.

Para poder hacer uso de la herramienta, el usuario deberá escribir un driver de EvoMaster, análogo al que se debe escribir para usar el cliente para Java. Dicho driver deberá ser implementado usando las librerías provistas por este paquete.

El siguiente es un ejemplo de driver escrito para una aplicación Flask que provee una API para crear, almacenar y borrar noticias denominada `news`:

```python
from evomaster_client.controller.flask_handler import FlaskHandler


class EMHandler(FlaskHandler):
    def package_prefixes_to_cover(self):
        """
        Define el listado de prefijos de paquetes que se quieren instrumentar.
        """
        return ['evomaster_benchmark.news']

    def flask_app(self):
        """
        Define el nombre del objeto correspondiente a la aplicación Flask.
        """
        return 'app'

    def flask_module(self):
        """
        Define el módulo donde se buscará el objeto Flask de la aplicación.
        """
        return 'evomaster_benchmark.news.app'

    def get_problem_info(self):
        return super().get_problem_info()

    def get_url(self):
        return super().get_url()

    def reset_state_of_sut(self):
```

```
30          """
31          Necesario para reiniciar el estado (sólo para aplicaciones
    ↪  stateful).
32          """
33          with self.server.app.app_context():
34              from evomaster_benchmark.news.model import db
35              db.drop_all()
36              db.create_all()
37
38      def setup_for_generated_test(self):
39          """
40          Opcional en caso de necesitar ejecutar una rutina antes de correr un
    ↪  test.
41          """
42          pass
43
44      def get_info_for_authentication(self):
45          """
46          Utilizado para autenticar requests a la API del sistema bajo
    ↪  testeo./
47          No soportado todavía por esta versión para Python.
48          """
49          return []
50
51      def get_preferred_output_format(self):
52          """
53          Define el lenguaje de salida de los tests generados.
54          """
55          return 'PYTHON_UNITTEST'
```

Luego, para levantar el driver podemos usar el CLI de la siguiente manera:

```
python -m evomaster_client.cli run-em-handler
    --handler-module 'evomaster_benchmark.em_handlers.news'
    --handler-class 'EMHandler'
    --instrumentation-level 2
```

Una vez ejecutado ese comando, el driver de Evomaster estará listo para recibir instrucciones del core y comenzar la generación de tests automática.

4. EVALUACIÓN

4.1. Benchmark

Para evaluar el funcionamiento del nuevo controlador para Python de EvoMaster se utilizó un benchmark con cinco sistemas bajo testeo diferentes. El benchmark se encarga de ejecutar el controlador de EvoMaster asociado a cada aplicación a testear, para luego ejecutar el core encargado de generar el código de los tests *white-box* correspondientes. En la siguiente sección analizaremos la cobertura obtenida en cada test suite para cada caso.

Algunos de los sistemas que utilizamos en el benchmark son artificiales, y han sido desarrollados en base a algunos de los benchmarks pre-existentes escritos para Java, disponibles en [4]. Estos benchmarks corresponden a tres APIs diferentes, y fueron implementados usando el framework Flask y el plugin Flask-RestX para desarrollo de APIs REST.

Los otros dos sistemas utilizados en el benchmark corresponden a aplicaciones públicas disponibles en GitHub que cumplen los requisitos necesarios para nuestro controlador. Dichas aplicaciones son basadas en Python y exponen una API Rest junto con su documentación OpenAPI.

El benchmark además está configurado para ser ejecutado con distintos parámetros que modifican el nivel de instrumentación realizada por el controlador de Python. De esta manera, podemos visualizar la diferencia obtenida con las distintas optimizaciones realizadas en el código del controlador. Definimos cuatro niveles de instrumentación de la siguiente manera:

- Nivel 0: Código sin instrumentación. Simula ejecuciones *black-box* donde no se tiene acceso al código fuente de la aplicación.
- Nivel 1: Instrumentación que sólo mide cobertura de líneas de código de la aplicación.
- Nivel 2: Instrumentación que además de medir cobertura, mide y hace seguimiento de los valores de *branch distance* obtenidos al hacer comparaciones dentro del código.
- Nivel 3: Instrumentación que además de medir cobertura, y *branch distance* de comparaciones, calcula y hace seguimiento de los valores de *branch distance* obtenidos al realizar operaciones booleanas de dos operandos dentro de flujos de control.

Veamos cómo difiere el código generado para el siguiente ejemplo en cada nivel de instrumentación definido:

Dado el siguiente código Python:

Código 1 ejemplo sin instrumentar

```
if x > 0 and y:
    return 1
else:
    return 0
```

se obtienen los siguientes códigos instrumentados:

Código 2 ejemplo de instrumentación nivel 1

```
if x > 0 and y:
    entering_statement('test.py', 2, 1)
    return completing_statement(1, 'test.py', 2, 1)
else:
    entering_statement('test.py', 4, 2)
    return completing_statement(0, 'test.py', 4, 2)
```

Código 3 ejemplo de instrumentación nivel 2

```
if compare_statement(x, '>', 0, 'test.py', 1, 1) and y:
    entering_statement('test.py', 2, 1)
    return completing_statement(1, 'test.py', 2, 1)
else:
    entering_statement('test.py', 4, 2)
    return completing_statement(0, 'test.py', 4, 2)
```

Código 4 ejemplo de instrumentación nivel 3

```
if and_statement(lambda : compare_statement(x, '>', 0, 'test.py', 1, 1),
    lambda: y, False, 'test.py', 1, 2):
    entering_statement('test.py', 2, 1)
    return completing_statement(1, 'test.py', 2, 1)
else:
    entering_statement('test.py', 4, 2)
    return completing_statement(0, 'test.py', 4, 2)
```

A continuación se describen brevemente las tres aplicaciones artificales desarrolladas para nuestro benchmark:

- NCS: API stateless que expone endpoints para ejecutar operaciones numéricas.
- SCS: API stateless que expone endpoints para ejecutar operaciones sobre cadenas de caracteres.
- NEWS: API con estado, que expone endpoints para manipular objetos del negocio relacionado con noticias.

Dichas aplicaciones fueron ideadas originalmente para testear el funcionamiento del controlador de EvoMaster para Java, y fueron traducidas a mano para llevar a cabo este benchmark sobre el nuevo controlador para Python Flask.

Por otro lado, tenemos las dos aplicaciones públicas que usaremos como casos de prueba. Estos sistemas presentan una complejidad más elevada que los sistemas artificiales, demostrando la robustez del nuevo controlador:

- Gilda: API que permite identificar entidades biomédicas aplicando desambiguación contextual. El código fuente puede encontrarse en [9].
- Flexget: API que permite el trackeo de torrents, nzbs, podcasts, comics, series, películas, mediante la interacción con otras APIs. El código fuente puede encontrarse en [8].

Benchmark App	**#Endpoints**				**#Líneas**
	GET	POST	PUT	DELETE	
ncs	6	0	0	0	242
scs	11	0	0	0	167
news	3	1	2	1	121
gilda	2	3	0	0	574
flexget	72	17	17	30	747

Tab. 4.1: Datos sobre cantidad de endpoints y líneas a cubrir para cada caso de estudio

4.2. Resultados

Para medir la performance de la herramienta en cada caso, observaremos el *coverage* o cobertura del código fuente de cada aplicación obtenido al ejecutar los tests generados por EvoMaster. La cobertura puede medirse en cobertura de líneas de código a cubrir, y cobertura de branches o ramas de código. Para cada caso de estudio y cada nivel de instrumentación se lanzaron 10 ejecuciones del algoritmo de generación en el core con semillas diferentes. Cada ejecución se realizó durante 10 minutos, y todos los resultados mostrados a continuación corresponden a promedios tomados entre los resultados obtenidos con las distintas semillas. Para la generación de valores aleatorios de semillas se utilizó la liberaría Python `random`[17].

Para medir el coverage obtenido con cada test suite generada, se hace uso nuevamente del controlador. Los tests son ejecutados contra una versión instrumentada de la aplicación en cuestión, de modo que registramos las mismas métricas de cobertura con las que se alimenta el algoritmo de búsqueda. En este caso, dichas métricas han sido exportadas a un archivo para luego ser procesadas y reportadas a continuación.

App/Nivel de Instrumentación	Nivel 0	Nivel 1	Nivel 2	Nivel 3
ncs	16.0	29.6	42.8	43.1
scs	14.0	22.7	109.6	108.3
news	12.7	14.5	14.0	13.9
gilda	7.8	9.5	12.0	13.8
flexget	207.0	214.1	216.0	214.2

Tab. 4.2: Número de tests generados por aplicación (promedio de 10 semillas de 10 minutos)

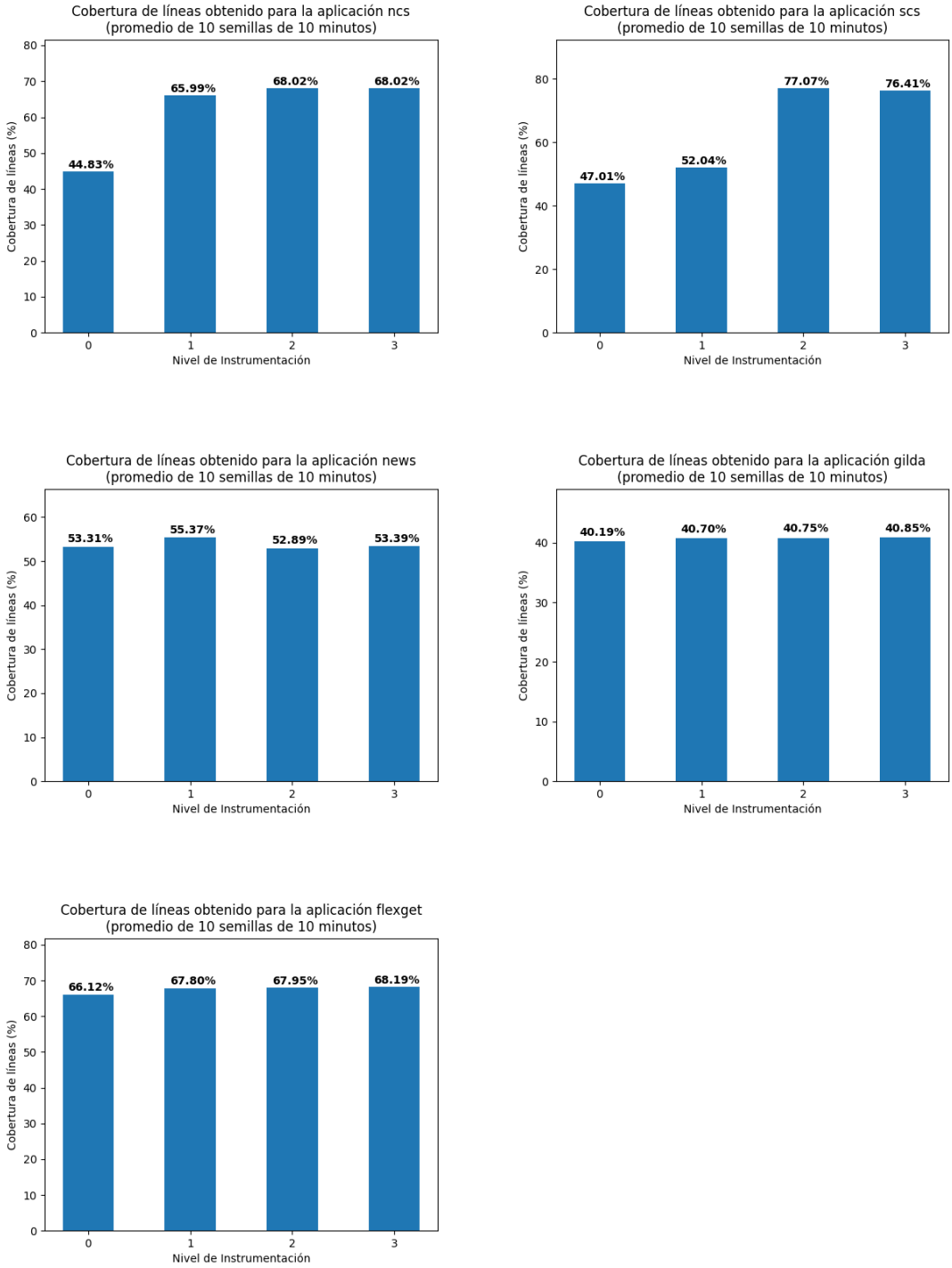

Fig. 4.1: Porcentaje de cobertura de líneas por aplicación para cada nivel de instrumentación

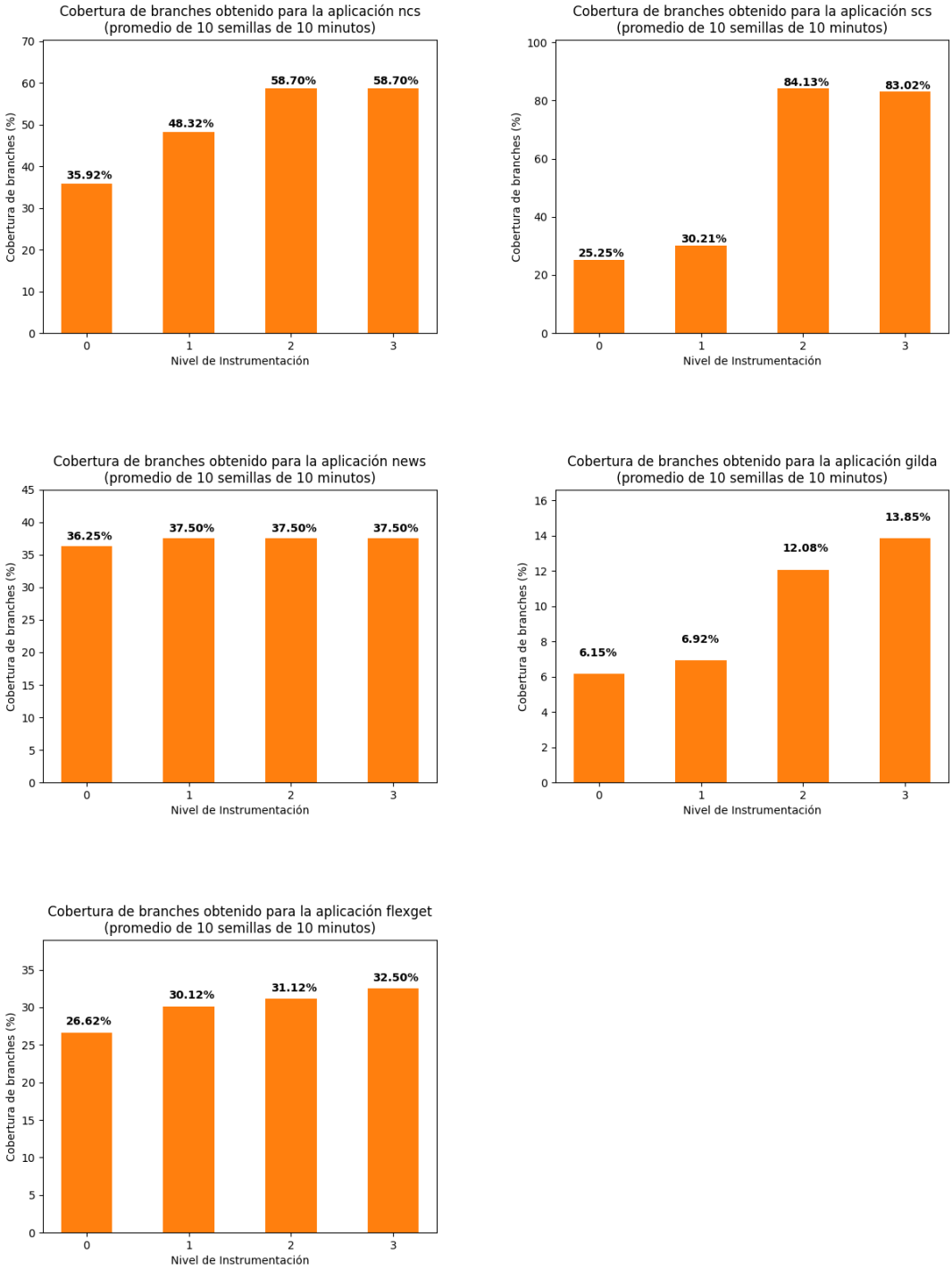

Fig. 4.2: Porcentaje de cobertura de branches por aplicación para cada nivel de instrumentación

Como primera observación, los resultados para las aplicaciones NCS y SCS muestran un patrón similar. A medida que el nivel de instrumentación aumenta, la misma se vuelve más compleja, generando métricas más ricas que permiten la generación de testeos más eficientes aumentando la cobertura de sentencias considerablemente. Es posible notar que el salto más grande se da entre los niveles de instrumentación 1 y 2, con la introducción de la noción de `branch distance` al instrumentador. Vemos también que el agregado de instrumentación para operaciones booleanas no es tan alto como el anterior mencionado. Debemos mencionar que el código de nuestros casos de estudios intentan maximizar la cantidad de flujos de control con distintas operaciones de comparación, con el fin de poder observar como difieren los resultados al enriquecer nuestra instrumentación. Es decir, cuanto más complejo sea el código a instrumentar, más información deberemos informar al core con las métricas del driver.

Si observamos los resultados para la aplicación NEWS, vemos valores completamente distintos. Aquí hay que notar que estamos hablando de una aplicación de muy baja complejidad, pero a diferencia de las otras dos, la aplicación NEWS cuenta con un estado. El código es sencillo y no tiene demasiados flujos de control, por lo cual la instrumentación no agrega valor a los testeos generados por EvoMaster. Este benchmark, por lo tanto, sirve más como una herramienta de validación de la capacidad de generación de testeos válidos para una aplicación con estado basada en Python Flask.

Es interesante notar que para la experimentación realizada con el controlador Java, cuyos resultados pueden encontrarse publicados en [2], la cobertura de líneas alcanzada para la aplicación NEWS fue de un 68 %, en comparación al 54 % obtenido con el controlador Python para la versión traducida de la aplicación. En general, los resultados de cobertura para los casos de estudio mencionados en dicho paper muestran valores entre un 32 % y 65 % de cobertura de líneas, lo cual hace que los resultados obtenidos con esta versión inicial de instrumentación Python se consideren prometedores.

Con respecto a las aplicaciones de terceros, se puede apreciar una mejora en la cobertura de branches a medida que aumentamos el nivel de instrumentación, como era de esperarse. A diferencia de las otras aplicaciones artificiales, la cobertura de líneas no muestra notables diferencias entre los distintos niveles.

Debemos observar que para los dos casos reales de aplicaciones públicas, debimos omitir ciertas partes del código que no han podido ser cubiertas por la versión actual del controlador presentada en este trabajo. Un ejemplo de las mismas son el código relacionado con autorización y autenticación de las APIs. Para poder ejecutar el algoritmo de generación de tests se modificaron las aplicaciones para remover el código que definía políticas de acceso a los distintos recursos expuestos, y todo el código relacionado con registro de usuarios de la aplicación.

Para el caso de `gilda`, se redujo el tamaño del dataset a partir del cual se realizaba la desambiguación de los datos, y se trabajó con una muestra reducida (de pocos megabytes) y descargada localmente al ejecutar la versión instrumentada del SUT.

5. CONCLUSIONES Y TRABAJO FUTURO

EvoMaster es una herramienta que permite la generación automática de testeos de sistema, y cuenta con un core lo suficientemente flexible para ser extendido a aún más lenguajes de programación y frameworks para generación de aplicaciones web REST. Este trabajo demuestra que una extensión para Python es posible, siempre y cuando el driver implemente el protocolo de comunicación defino por el core y provea de las métricas necesarias para la evolución de los tests.

Cada driver puede proveer un subconjunto de las métricas soportadas por el core. Para el driver de Python, no se contemplaron algunos features soportados por EvoMaster como el manejo de comunicaciones SQL para generar e inyectar datos automáticamente en las bases de datos de los sistemas bajo testeo, o la autenticación de requests basada en headers y cookies.

Como trabajo futuro, el driver de EvoMaster para Python puede ser extendido para otros frameworks más complejos que Flask, como lo es Django, con el objetivo de poder ampliar el alcance de aplicaciones instrumentables. Además, la instrumentación implementada hasta el momento se puede volver más compleja para poder generar mayor cobertura en aplicaciones más grandes. Un ejemplo de esto es extender la instrumentación para expresiones con más de dos operandos.

El testing es una parte fundamental del desarrollo de software, aunque a veces puede volverse una tarea costosa en términos de tiempo y esfuerzo. Considero que este tipo de herramientas tiene el potencial de mejorar la manera en que como desarrolladores testeamos nuestras aplicaciones y mejoramos la calidad de las mismas haciendo uso de heurísticas y algoritmos basados en inteligencia artificial.

Bibliografía

[1] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 205–214, 2010.

[2] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1), January 2019.

[3] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliab.*, 24(3):219–250, May 2014.

[4] Repositorio de benchmarks para evomaster. `https://github.com/EMResearch/EMB`.

[5] Repositorio de evomaster. `https://github.com/EMResearch/EvoMaster`.

[6] Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, 2000. AAI9980887.

[7] Flask. `https://palletsprojects.com/p/flask`.

[8] Repositorio de aplicación flexget. `https://github.com/Flexget/Flexget`.

[9] Repositorio de aplicación gilda. `https://github.com/indralab/gilda`.

[10] Java. `https://www.java.com/`.

[11] Javascript. `https://www.javascript.com/`.

[12] Pep 302 – new import hooks. `https://www.python.org/dev/peps/pep-0302/#id21`.

[13] Pypi. `https://pypi.org/`.

[14] Pytest. `https://pytest.org/`.

[15] Pytest coverage plugin. `http://pytest-cov.rtfd.org/`.

[16] Python. `https://www.python.org/`.

[17] Generate pseudo-random numbers. `https://docs.python.org/3/library/random.html`.

[18] Swagger openapi. `https://swagger.io/resources/open-api/`.