



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

EPAs para REST APIs con EvoMaster

Tesis de Licenciatura en Ciencias de la Computación

Chiara Tarzia

Director: Dr. Juan P. Galeotti

Buenos Aires, 2024

EPAS PARA REST APIS CON EVOMASTER

La validación y verificación de las REST APIs puede resultar altamente compleja debido a la gran cantidad de endpoints que pueden contener. Cada combinación de llamados HTTP puede requerir diversas precondiciones que pueden no estar documentadas. En este trabajo, se propone una innovadora solución para la generación de Enabledness-Preserving Abstractions (EPAs) para REST APIs utilizando EvoMaster, herramienta de generación de tests automatizados con algoritmos evolutivos. Las EPAs son una representación del modelo mental que los desarrolladores tienen de una API, modelando el comportamiento del código y facilitando su análisis y comprensión.

La propuesta se destaca por su enfoque dinámico en la generación de EPAs, a diferencia de los métodos estáticos tradicionales.

Adicionalmente, se plantea la modificación del algoritmo evolutivo MIO de EvoMaster, incorporando una heurística que prioriza la identificación de nuevos arcos en las EPAs. Esta estrategia permite enfocar los casos de test para maximizar de la cobertura de la EPA.

Palabras claves: REST APIs, EvoMaster, EPA, algoritmos genéticos, MIO, validación, verificación, tests automatizados.

EPAS FOR REST APIS WITH EVOMASTER

Validation and verification of REST APIs can be highly complex due to the large number of endpoints they can contain. Each combination of HTTP calls may require various preconditions that may not be documented. This thesis proposes an innovative solution for generating Enabledness-Preserving Abstractions (EPAs) for REST APIs using EvoMaster, a tool for generating automated tests with evolutionary algorithms. EPAs are a representation of the developers' mental model of an API, modeling the behavior of the code and facilitating its analysis and understanding.

The proposal stands out for its dynamic approach to EPA generation, in contrast to traditional static methods. This feature allows for a more accurate representation of the API's behaviour in real time.

Additionally, a modification to the MIO evolutionary algorithm of EvoMaster is proposed, incorporating a heuristic that prioritizes the identification of new edges in the EPAs. This strategy allows focusing the test cases to maximise the coverage of the EPA, optimising the validation process.

In summary, this work presents a valuable contribution to the field of REST API validation and verification. The dynamic generation of EPAs and the implementation of the new heuristic in EvoMaster make up a powerful tool that facilitates error detection and quality improvement of APIs.

Keywords: REST APIs, EvoMaster, EPA, genetic algorithms, MIO, validation, verification, automated tests.

AGRADECIMIENTOS

Gracias mamá y papá por todo.

Gracias familia.

Gracias amigos de la facultad y de la vida.

Gracias JP por la paciencia y la ayuda en esta tesis.

Gracias Taylor por musicalizar mientras escribía.

Gracias Eli por acompañarme mientras estudiaba para todos los parciales, recuperatorios y finales de la carrera, y mientras escribía esta tesis.

Índice general

1..	Introducción	1
2..	Preliminares	3
2.1.	REST APIs	3
2.2.	EvoMaster	6
2.2.1.	Funcionamiento de EvoMaster	6
2.2.2.	Algoritmo MIO	7
2.3.	Enabledness-Preserving Abstractions (EPAs)	8
3..	Desarrollo	12
3.1.	Generación de EPA con EvoMaster	12
3.2.	Nueva heurística para EPAs en EvoMaster	16
3.3.	Discusión	16
4..	Experimentación	18
4.1.	REST APIs utilizadas en la experimentación	18
4.1.1.	Employees	18
4.1.2.	Features Service	19
4.1.3.	Proxyprint	19
4.2.	Resultados	23
5..	Trabajo futuro	32
6..	Conclusiones	33

1. INTRODUCCIÓN

Las REST APIs (REpresentational State Transfer Application Programming Interface) son una parte fundamental de las aplicaciones modernas y sistemas distribuidos. A medida que la complejidad de estas interfaces aumenta, también crece la posibilidad de que estas tengan errores que causen el mal funcionamiento de las mismas. Es vital encontrar y remediar estos errores de manera rápida y efectiva, especialmente cuando sobre estas APIs se construyen sistemas que necesitan ser resilientes y correctos como aplicaciones bancarias o sistemas médicos.

La validación y verificación de REST APIs presenta múltiples desafíos. Para este trabajo, lo que mas nos afecta es que las REST APIs pueden tener una gran cantidad de endpoints, y cada combinación de estos puede tener diferentes comportamientos y precondiciones. Estas últimas no son necesariamente claras, y pueden no estar documentadas. Por esto, hacer un testeo exhaustivo puede ser muy costoso.

Estos sistemas se pueden validar y verificar con tests escritos por personas, pero estos son afectados por limitaciones de tiempo y la subjetividad de los desarrolladores. Es por esto que es importante complementar los tests manuales de las APIs con testeo automatizado para garantizar su correcto funcionamiento. EvoMaster es una herramienta *open-source* para generar tests automatizados para APIs utilizando algoritmos evolutivos que ofrece una solución a estos desafíos.

Las Enabledness-Preserving Abstractions (EPAs) son una forma de representar el modelo mental que tiene el desarrollador de una API. Son abstracciones que modelan el comportamiento del código y se pueden usar para validarlo e identificar fallas en el código o la documentación sobre los requerimientos de cada acción.

El objetivo principal de esta tesis es extender EvoMaster con la funcionalidad de generar una EPA basada en la batería de tests generada por la herramienta. Esta EPA no necesariamente representará toda la REST API que se está testeando, es una subaproximación, solamente reflejará el comportamiento observado en los tests. De todos modos, será una herramienta mas que permitirá al desarrollador de la API entender mejor su aplicación y encontrar fallas en la misma, enriqueciendo la experiencia de los usuarios de EvoMaster para testear sus REST APIs. Esta propuesta es especialmente relevante porque plantea generar dinámicamente la EPA, a diferencia de los enfoques estáticos que prevalecen actualmente.

También se modificará al algoritmo evolutivo MIO de EvoMaster para que le de valor a encontrar nuevos arcos de la EPA. Esta nueva heurística ayudará a priorizar los casos de test que aportan nuevos caminos, para construir una EPA lo mas completa posible.

Por último se realizará una experimentación para evaluar el funcionamiento de estas nuevas adiciones. Se presentará su utilidad para encontrar desperfectos en las aplicaciones y se mostrarán ejemplos de su uso en REST APIs reales. También se medirá la efectividad de la nueva heurística en el algoritmo para generar EPAs mas “completas”.

Podemos resumir las contribuciones de este trabajo como:

- Implementación en EvoMaster de generación de una subaproximación de una EPA.
- Implementación en EvoMaster de la heurística para generación de EPAs.

-
- Evaluación de esta implementación con REST APIs reales y generadas para este trabajo.

El resto del trabajo se organiza de la siguiente manera. Primero se presentará marco teórico de la tesis en el capítulo 2. A continuación, en el capítulo 3 se detallará cómo funciona el algoritmo de generación de EPAs y su heurística correspondiente en EvoMaster. En el capítulo 4 se presentará la experimentación en la que evaluaremos los aportes de esta tesis. Luego se presentarán posibles trabajos a futuro en la sección 5. Por último, en el capítulo 6 veremos las conclusiones.

2. PRELIMINARES

Antes de presentar la explicación detallada de la generación de EPAs para REST APIs con EvoMaster, es esencial establecer una comprensión sólida de algunos conceptos fundamentales. Estos son el marco teórico necesario para comprender la contribución de esta investigación.

A continuación, se presentan tres conceptos clave que serán recurrentes a lo largo de esta tesis:

2.1. REST APIs

En pocas palabras, una API es un mecanismo que permite a una aplicación (cliente) acceder a un recurso de otra aplicación (servidor). Las REST APIs tienen que acatar los seis principios de diseño REST [1]:

- **Interfaz uniforme:** Todas las llamadas al mismo recurso tienen que tener la misma estructura, sin importar su origen.
- **Desacoplamiento cliente-servidor:** Las aplicaciones de cliente y servidor tienen que ser completamente independientes una de la otra.
- **Sin estados:** Cada llamada a un recurso necesita incluir toda la información necesaria para procesarla.
- **Posibilidad de caché:** Cuando sea posible, los recursos deben ser almacenables en caché tanto en el lado del cliente como en el lado del servidor. Esto le da mejor rendimiento al cliente y escalabilidad al servidor.
- **Arquitectura en capas:** Las llamadas HTTP pueden pasar por varias capas. El cliente y servidor no necesariamente se conectan entre si directamente.
- **Código a demanda (opcional):** Las REST APIs en algunos casos pueden contener código ejecutable. Este solo debe ejecutarse a demanda.

Las REST APIs se comunican a través de llamados HTTP. Cada uno de estos llamados incluye un verbo que está típicamente asociado con un tipo de operación. Por ejemplo, *POST* crea un registro, *GET* lo devuelve, *PUT* lo actualiza y *DELETE* lo elimina.

Cuando la API retorna la información que tiene sobre un recurso, esta puede estar en varios formatos. Algunos de los mas populares son JSON, HTML, XLT o texto plano. Esta respuesta también incluye un código HTTP, el cual tiene un significado específico por convención. Para ilustrar, 100 significa “continúe”, 202 representa “aceptado” y 404 es el caso de “no encontrado”.

Un llamado a la REST API también incluye encabezados y parámetros que contienen información importante para la operación. Tal como metadata, autorizaciones, URIs y cookies.

Todas estas son características de una REST API bien diseñada.

A modo de ejemplo, **employees** es una REST API simple que contiene solamente operaciones CRUD: *Create*, *Read*, *Update*, *Delete* para registros del tipo **Employee**.


```
1 @RestController
2 public class EmployeeController {
3     // Employee = {id, firstName, lastName, email}
4     List<Employee> employees = new ArrayList<>();
5     private long nextId = 0;
6
7     // get all employees
8     @GetMapping("/employees")
9     public ResponseEntity<List<Employee>> getAllEmployees(){
10         return ResponseEntity.ok(employees);
11     }
12
13     // create employee
14     @PostMapping("/employees")
15     public ResponseEntity<Employee> createEmployee(@RequestBody Employee
16         employee) {
17         employee.setId(nextId);
18         nextId++;
19         employees.add(employee);
20         return ResponseEntity.ok(employee);
21     }
22
23     // get employee by id
24     @GetMapping("/employees/{id}")
25     public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {
26         Employee employee = getEmployee(id);
27         return ResponseEntity.ok(employee);
28     }
29
30     // update employee
31     @PutMapping("/employees/{id}")
32     public ResponseEntity<Employee> updateEmployee(@PathVariable Long id,
33         @RequestBody Employee employeeDetails){
34         Employee employee = getEmployee(id);
35
36         employee.setFirstName(employeeDetails.getFirstName());
37         employee.setLastName(employeeDetails.getLastName());
38         employee.setEmailId(employeeDetails.getEmailId());
39
40         return ResponseEntity.ok(employee);
41     }
42
43     // delete employee
44     @DeleteMapping("/employees/{id}")
45     public ResponseEntity<Map<String, Boolean>> deleteEmployee(@PathVariable
46         Long id){
47         Employee employee = getEmployee(id);
48         employees.remove(employee);
49
50         Map<String, Boolean> response = new HashMap<>();
51         response.put("deleted", Boolean.TRUE);
52         return ResponseEntity.ok(response);
53     }
54
55     // delete all employees
56     @DeleteMapping("/employees")
57     public ResponseEntity<Map<String, Boolean>> deleteAllEmployees(){
58         employees = new ArrayList<>();
59     }
```

```
56
57     Map<String, Boolean> response = new HashMap<>();
58     response.put("deleted", Boolean.TRUE);
59     return ResponseEntity.ok(response);
60 }
61
62 private Employee getEmployee(Long id) {
63     Optional<Employee> employee = employees.stream().filter(e -> id.equals(
64         e.getId())).findFirst();
65     if (!employee.isPresent()) {
66         throw new ResourceNotFoundException("Employee not exist with id : " +
67             id);
68     }
69     return employee.get();
70 }
```

Extracto de código 2.1: Ejemplo de REST API.

Veamos como nuestra API CRUD `employees` cumple con los principios de diseño REST:

- **Basada en recursos:** Las REST y CRUD APIs son basadas en recursos, es decir, todos los elementos con los que interactúan son recursos. En este caso, nuestro recurso es `employee` y tiene un identificador único que lo caracteriza(`id`).
- **Interfaz uniforme:** Todas las llamadas al mismo recurso tienen la misma estructura siguiendo los verbos HTTP:
 - *POST - CREATE:* Crea un nuevo recurso.
 - *GET - READ:* Responde con la representación del recurso.
 - *PUT o PATCH - UPDATE:* Modifica recursos existentes.
 - *DELETE - DELETE:* Elimina recursos.
- **Desacoplamiento cliente-servidor:** `employees` expone sus endpoints, los cuales los clientes pueden llamar para realizar operaciones CRUD. Ninguno depende del otro.
- **Sin estados:** Cada llamado incluye toda la información para aplicar la operación. Por ejemplo, el `id` del `employee` que se quiere obtener.
- **Posibilidad de caché:** Se podría implementar una caché en la que se almacenan los últimos `employees` que se leyeron. Esto tendría sentido en el caso de que también se implemente una base de datos.
- **Arquitectura en capas:** Los clientes interactúan con los endpoints de `employees` sin necesidad de entender como funciona por dentro. No ocurre en este caso pero podría tener más capas, y que se llame a otros recursos desde la API.
- **Código a demanda:** No aplica en este caso.

2.2. EvoMaster

EvoMaster [2] [3] es una herramienta *open-source* para generar tests automatizados para APIs utilizando algoritmos evolutivos. Este encuentra inputs que hacen fallar a la aplicación y genera una batería de tests en formato JUnit.

La herramienta funciona con el algoritmo evolutivo MIO (Many Independent Objective) [4], junto con análisis dinámico de programas. Para generar una batería de tests efectiva, comienza su análisis con casos de test aleatorios que luego evoluciona. Se mide la efectividad de cada test en base a su cobertura de código, errores y nuevas respuestas HTTP recibidas, entre otras heurísticas. EvoMaster también tiene en cuenta las buenas prácticas de diseño de REST APIs para generar tests de manera mas eficiente.

EvoMaster puede generar casos de test para APIs REST, GraphQL y RPC. Estos pueden ser de tipo caja negra (sin conocimiento de la implementación de la aplicación) o caja blanca (con conocimiento completo de la implementación, incluyendo el código fuente). En el primer caso, la herramienta funciona para APIs implementadas en cualquier lenguaje de programación. En el segundo, solo para APIs compiladas a JVM (*e.g.* Java y Kotlin). En el modo de testeo de caja blanca se obtienen mejores resultados, ya que puede analizar el código y aplicar una mayor cantidad de heurísticas.

Es importante destacar que evidencia experimental muestra que EvoMaster tiene uno de los mejores desempeños en comparación a otros *fuzzers* para REST APIs [5].

Para esta tesis nos enfocaremos en el caso de uso particular de EvoMaster para REST APIs de caja blanca. Para poder utilizar la herramienta en este caso debemos implementar para la REST API a testear, o *System Under Test* (SUT), un driver que permitirá que EvoMaster interactúe con la misma. Este especificará cómo iniciar y frenar el SUT, reiniciar su estado, entre otras operaciones. También con este driver se le debe informar a EvoMaster cómo es el esquema de la API en formato OpenAPI/Swagger.

Al finalizar, EvoMaster retornará una batería de tests JUnit escritos en Java o Kotlin. Algunos de estos tests pueden mostrar errores en la aplicación y otros pueden mostrar una lista de llamados a la API que son exitosos.

2.2.1. Funcionamiento de EvoMaster

Para finalizar esta sección, resumiremos los pasos que realiza EvoMaster para generar una suite de test para el SUT. No repasaremos todos los pasos, pero si los mas relevantes para comprender los aportes de esta tesis.

Mientras el driver del SUT está ejecutandose para la API a testear, iniciamos el programa EvoMaster. En primer lugar, EvoMaster se comunica con el driver y determina qué clase de API es la que se va a testear, las bases de datos que utiliza, cómo inicializarla y su esquema en formato OpenAPI/Swagger.

Los casos de test mientras están en “construcción”, no son mas que una lista de llamados HTTP a la API. La generación de la batería de tests la realiza el algoritmo MIO que se explicará en detalle en la sección 2.2.2.

Para ciertos casos de test, EvoMaster puede recurrir a sumar información a la base de datos comunicándose directamente con la misma. Esto permite obtener configuraciones de la aplicación que pueden ser difíciles o imposibles de alcanzar. Este comportamiento puede ser activado y desactivado utilizando configuraciones de EvoMaster. En el caso de base de datos SQL tenemos las configuraciones `generateSqlDataWithSearch` y `generateSqlDataWithDSE`.

Para cada uno de estos tests que tiene almacenado, se guarda la lista de pasos que forman este test. Para cada método se guarda información sobre cada llamado HTTP al SUT, incluyendo el código HTTP que respondió la aplicación, el *body* y los mensajes de error. Esta es parte de la información que se utiliza para medir la cobertura del caso de test y aplicar heurísticas de búsqueda.

Al finalizar la búsqueda, limitada únicamente por el tiempo configurado por el usuario, los tests generados se traducen a tests de JUnit y se graban en la carpeta configurada por el usuario.

2.2.2. Algoritmo MIO

MIO (Many Independent Objective) [4] es el algoritmo evolutivo que utiliza EvoMaster. Su función es maximizar la cobertura de los múltiples *targets* que tiene como objetivo con los tests generados. Un ejemplo de *target* puede ser que el llamado a cierto endpoint responda el código HTTP 200.

La particularidad que tiene el problema de generación de tests es que el algoritmo tiene que balancear adicionar más tests para cubrir nuevos targets, junto con el objetivo secundario de cubrir la mayor cantidad de targets con la menor cantidad de tests cortos como sea posible. Otra dificultad es que la cantidad de targets puede estar en el orden de las decenas de miles, especialmente para APIs de escala industrial. Optimizar para esa cantidad de objetivos puede ser altamente desafiante. MIO ha sido diseñado específicamente para este caso de uso, y tiene un excelente desempeño.

A continuación se presentará una exposición detallada de los pasos del funcionamiento de MIO.

MIO mantiene un archivo de tests, en el que para cada target se almacena una población de tests de tamaño hasta n . Si se tienen z targets, puede haber hasta $n \times z$ tests en el archivo a la vez.

En la primera búsqueda el archivo está vacío, por lo que genera un nuevo test aleatorio. Luego MIO decide entre generar un nuevo test aleatorio (probabilidad P_r), o copiar un test del archivo y mutarlo (probabilidad $1 - P_r$). En ambos casos, se calcula su fitness y se guarda una copia del test en cero o mas de las z poblaciones del archivo. Para cada target se asigna un puntaje $h \in [0, 1]$. Si el puntaje es 1 significa que ese target esta cubierto, si es 0 es el menor posible valor de la heurística.

Para cada target k , se guarda un test en la población T_k ($|T_k| \leq n$) si $h_k > 0$ y se cumple alguna de las siguientes condiciones:

- Si el target esta cubierto ($h_k = 1$) el test se agrega y se descartan todos los otros tests de la población, manteniendo su tamaño durante el resto de la búsqueda en $|T_K| = 1$. El test solo será reemplazado si:
 - Se encuentra otro test más corto. En este caso, tiene una menor cantidad de llamados HTTP, o
 - Si el nuevo test tiene mejor cobertura de otros targets.
- Si la población no está llena ($|T_K| < n$) el test se agrega. En caso contrario ($|T_K| = n$), reemplaza al peor test de la población si y solo si:
 - El nuevo test tiene mejor valor de heurística, o

- Si ambos tienen el mismo h pero el tamaño del nuevo test no es mayor.

Cuando MIO elige mutar un test del archivo, entonces realiza los siguientes pasos:

1. Elige un target k aleatorio con $|T_k| > 0$ que no este cubierto. Si todas las poblaciones no vacías están cubiertas, elegir una aleatoria entre ellas.
2. Elige un test aleatorio de T_k .

A continuación en el algoritmo 1 se presenta el pseudocódigo de MIO [6].

Algoritmo 1 Pseudocódigo de MIO

Input: Condición de frenado C , Función de *fitness* δ , Tamaño de población n , Probabilidad de generar un nuevo test aleatorio P_r , Inicio de búsqueda F

Output: Archivo optimizado de individuos A

```

1:  $T \leftarrow \text{SETOFEMPTYPOPULATIONS}()$ 
2:  $A \leftarrow \{\}$ 
3: while  $\neg C$  do
4:   if  $P_r > \text{rand}()$  then
5:      $p \leftarrow \text{RANDOMINDIVIDUAL}()$ 
6:   else
7:      $p \leftarrow \text{SAMPLEINDIVIDUAL}(T)$ 
8:      $p \leftarrow \text{MUTATE}(p)$ 
9:   end if
10:  for all  $k \in \text{REACHEDTARGETS}(p)$  do
11:    if  $\text{ISTARGETCOVERED}(k)$  then
12:       $\text{UPDATEARCHIVE}(A, p)$ 
13:       $T \leftarrow T \setminus \{T_k\}$ 
14:    else
15:       $T_k \leftarrow T_k \cup \{p\}$ 
16:      if  $|T_k| > n$  then
17:         $\text{REMOVETWORSTTEST}(T_k, \delta)$ 
18:      end if
19:    end if
20:  end for
21:   $\text{UPDATEPARAMETERS}(F, P_r, n)$ 
22: end while
23: return  $A$ 

```

2.3. Enabledness-Preserving Abstractions (EPAs)

Las EPAs son una manera de representar aplicaciones con requerimientos no triviales para ordenar sus métodos, como las APIs [7]. Formalizan el “modelo mental” que tiene el desarrollador de una API. Son abstracciones que modelan el comportamiento del código, que se pueden usar para validarlo e identificar fallas en el mismo o la documentación sobre los requerimientos de cada acción.

Podemos representar una clase como un LTS (*Labelled Transition System*), una máquina de estados infinita y determinística. Esta tiene un estado inicial y cada acción nos lleva

a otro estado. Por ejemplo, supongamos que tenemos una implementación de una lista. Comenzamos con una lista vacía como estado inicial. Se le pueden agregar y quitar elementos de a uno por vez las veces que queramos. También se puede destruir la lista. Este caso puede potencialmente necesitar un espacio de estados infinito para ser representado. En la figura 2.1 vemos plasmado este ejemplo. [7]

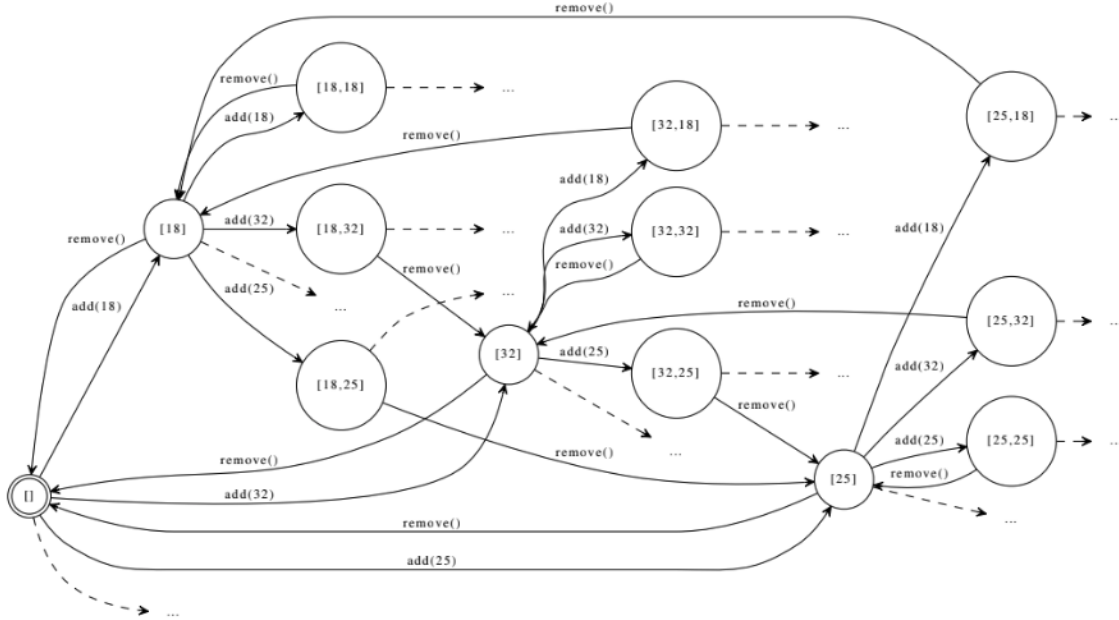


Fig. 2.1: Fragmento finito del LTS de una lista [7].

Para poder representar la misma información con una máquina de estados no determinística en un espacio finito, tenemos las EPAs. En esta abstracción se agrupan los estados del LTS según las operaciones que están habilitadas en cada uno. La EPA permite simular cualquier camino de la máquina de estados anterior.

Podemos ver la EPA que abstrae el LTS de la figura 2.1 en la figura 2.2. Los estados iniciales se marcan con un círculo. Podemos ver como con un solo estado (S7) representamos todos los estados en los que una lista tiene uno o más elementos, ya que habilitan las mismas operaciones: **add**, **remove**, **destroy**. En cambio, si la lista tiene solo un elemento (S5) solo podemos agregar un elemento o destruir la lista. También vemos como la implementación tiene errores, ya que hay casos en el que **add** causa que no se puedan agregar o quitar elementos y que no se pueda destruir la lista.

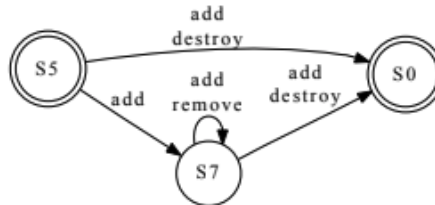


Fig. 2.2: EPA de una lista enlazada simple [7].

A continuación vamos a formalizar los conceptos presentados en esta sección. Esta

formalización fue presentada por de Caso et al. en [7].

Definición 2.3.1 (*Configuración*): Una configuración es el estado interno de la API y los elementos del *heap* (memoria). Se el conjunto de todas las configuraciones posibles con \mathbb{C} .

Definición 2.3.2 (*Action System*): Un *Action System* es la interpretación semántica del código fuente de un programa. Lo podemos representar con una estructura de la forma $AS = \langle Act, F, R, inv, init \rangle$, donde:

- $Act = \{a_1, \dots, a_n\}$ es el conjunto finito de etiquetas de acciones.
- F es el conjunto de funciones indexadas por Act . Para cada etiqueta a , la función $F_a : \mathbb{C} \times \mathbb{Z} \rightarrow (\mathbb{C} \cup \perp)$ toma una configuración y un parámetro entero (lo restringimos a esto sin perder generalidad) y:
 1. transforma la configuración, o
 2. no termina (\perp).
- R es el conjunto de cláusulas requeridas indexadas por Act . Para cada etiqueta a , la cláusula requerida $R_a : \mathbb{C} \times \mathbb{Z} \rightarrow \{true, false\}$ indica si la acción a esta habilitada para la configuración y parámetro indicados.
- $inv : \mathbb{C} \rightarrow \{true, false\}$ es el invariante del action system.
- $init : \mathbb{C} \rightarrow \{true, false\}$ es la condición inicial, indica si la configuración es una configuración inicial.

Definición 2.3.3 (*Predicado de un Action Set*): Dado un action system $AS = \langle Act, F, R, inv, init \rangle$, el predicado de un conjunto de acciones $A \subseteq Act$ es la función $pred_A : \mathbb{C} \rightarrow \{true, false\}$, definida como:

$$pred_A(c) \Leftrightarrow inv(c) \wedge \bigwedge_{a \in A} \exists p. R_a(c, p) \wedge \bigwedge_{a \notin A} \neg p. R_a(c, p)$$

En otras palabras, nos indica si una configuración tiene exactamente las mismas acciones de A habilitadas.

Definición 2.3.4 (*Enabledness-preserving Abstraction*): Dado un action system $AS = \langle Act, F, R, inv, init \rangle$, entonces $M = \langle \Sigma, S, S_0, \delta \rangle$ es la EPA de AS , donde:

1. $\Sigma = Act$
2. $S = 2^{Act}$
3. $S_0 = \{A \in S \mid \exists c \in \mathbb{C}. pred_A(c) \wedge init(c)\}$
4. Para cada $A \in S$ y $a \in \Sigma$, si $a \notin A$ entonces $\delta(A, a) = \emptyset$, de lo contrario:

$$\delta(A, a) = \left\{ B \mid \exists c. pred_A(c) \wedge \exists p. R_a(c, p) \wedge pred_B(F_a(c, p)) \right\}$$

En el paper [7] de Caso et al. propusieron un algoritmo que genera la EPA estáticamente para artefactos de código que tienen cláusulas requeridas, es decir, las precondiciones con los que tiene que cumplir el estado concreto de la aplicación para que sea válido llamar a cada uno de los métodos. De esta manera se puede representar el programa con una máquina de estados basado en las operaciones habilitadas (porque se cumplen sus precondiciones) para cada uno de los estados.

También implementaron este algoritmo para la herramienta *CONTRACTOR* que analiza programas en *C*. Este toma un programa con cláusulas requeridas y produce una EPA.

3. DESARROLLO

En esta sección se detallará el funcionamiento de las dos extensiones a EvoMaster presentadas en esta tesis. Primero veremos la implementación de la generación de EPAs y luego cómo funciona la nueva heurística.

3.1. Generación de EPA con EvoMaster

Supongamos que queremos utilizar EvoMaster para generar tests y una EPA para **employees**, la REST API CRUD presentada en la sección 2.1. **employees** será nuestro ejemplo de SUT para el resto de la explicación de los aportes de esta tesis a EvoMaster.

En primer lugar, para extender EvoMaster para generar EPAs, previamente debemos incorporar a nuestra REST API información de manera análoga a lo que se realiza en de Caso et al. [7] con cláusulas requeridas para la herramienta *CONTRACTOR*.

Para que EvoMaster pueda construir la EPA necesita saber que endpoints están habilitados en la API antes y después de ejecutar cada llamado HTTP. En lugar de crear una cláusula requerida para cada método, implementaremos un solo endpoint que le brinde a EvoMaster esta información con el verbo HTTP GET y la ruta `/enabledEndpoints`.

Este debe responder en base al estado actual de la API, qué endpoints están habilitados. Definimos como método habilitado a aquellos que con la correcta combinación de parámetros pueden ser llamados exitosamente. Es decir, esos métodos para los cuales se cumplen las precondiciones para ser llamados. A diferencia de la implementación de de Caso et al., nuestra implementación de las cláusulas requeridas no tiene en cuenta los parámetros de los llamados HTTP. Nos indicará qué métodos están habilitados únicamente basado en la configuración interna de la API.

En el caso de nuestra API **employees**, `GET::/enabledEndpoints` tiene que retornar:

- Si no hay empleados:
 - `GET::/employees`
 - `DELETE::/employees`

EMPLOYEE	
GET	<ul style="list-style-type: none">• <code>/employees</code>: obtiene todos los employee• <code>/employees/{id}</code>: obtiene employee por id
POST	<ul style="list-style-type: none">• <code>/employees</code>: crea employee
PUT	<ul style="list-style-type: none">• <code>/employees/{id}</code>: actualiza employee
DELETE	<ul style="list-style-type: none">• <code>/employees</code>: elimina todos los employees• <code>/employees/{id}</code>: elimina employee por id

Fig. 3.1: Endpoints de **employees**.

- POST::`/employees`
- Si hay uno o más empleados:
 - GET::`/employees`
 - DELETE::`/employees`
 - POST::`/employees`
 - GET::`/employees/{id}`
 - PUT::`/employees/{id}`
 - DELETE::`/employees/{id}`

Esto es porque GET::`/employees`, DELETE::`/employees` y POST::`/employees` están siempre habilitados. En cambio, GET::`/employees/{id}`, PUT::`/employees/{id}` y DELETE::`/employees/{id}` requieren poder apuntar a un id válido y existente. De otra manera, la API debería responder con un código HTTP de error.

```

1 @GetMapping("/enabledEndpoints")
2 public RestActionsDto getEnabledEndpoints() {
3     RestActionsDto restActionsDto = new RestActionsDto();
4
5     restActionsDto.enabledRestActions
6         .add(new RestActionDto("get", "/employees"));
7     restActionsDto.enabledRestActions
8         .add(new RestActionDto("delete", "/employees"));
9     restActionsDto.enabledRestActions
10        .add(new RestActionDto("post", "/employees"));
11
12     if (employees.size() > 0) {
13         restActionsDto.enabledRestActions
14             .add(new RestActionDto("get", "/employees/{id}"));
15         restActionsDto.enabledRestActions
16             .add(new RestActionDto("put", "/employees/{id}"));
17         restActionsDto.enabledRestActions
18             .add(new RestActionDto("delete", "/employees/{id}"));
19     }
20
21     return restActionsDto;
22 }
23

```

Extracto de código 3.1: Implementación de `/enabledEndpoints` para `employees`.

Es importante resaltar que estas precondiciones e implementación de GET::`/enabledEndpoints` son desarrolladas por el usuario y pueden tener errores. Al ver la EPA generada, el usuario puede potencialmente ver estos errores y corregir su implementación, de la misma manera que corregiría el código de su REST API.

A continuación, debemos implementar el driver `EmbeddedSutController` de la manera tradicional para `EvoMaster`. Para mayor detalle de los pasos a seguir, consultar la documentación de `EvoMaster` [8]. A la hora de implementar el método `getProblemInfo()` debemos tener en cuenta que el endpoint GET::`/enabledEndpoints` no es parte de la API `employees` en sí, y no nos interesa que se testeé como uno, por lo que lo agregaremos a la lista de endpoints a ignorar.

```

1  @Override
2  public ProblemInfo getProblemInfo() {
3      return new RestProblem(
4          "http://localhost:" + getSubPort() + "/v2/api-docs",
5          Arrays.asList("/enabledEndpoints",
6                      "/error"));
7  }

```

Extracto de código 3.2: Implementación de `getProblemInfo()` en `EmbeddedSutController` para `employees`.

En cuanto a cambios que se hicieron en `EvoMaster` para que pueda crear una EPA, en primer lugar se incorporó la configuración `epaCalculation`. Si esta está en `true`, se crea una EPA para la REST API que se está testeando. También con la configuración `epaFile` el usuario puede especificar dónde y bajo qué nombre se debe guardar la EPA.

Luego se le agregó a `RestCallResult` la posibilidad de guardar dos grupos de endpoints habilitados para ese llamado HTTP. Primero, los endpoints habilitados antes del llamado HTTP (`enabledEndpointsBeforeAction`). Este solo tendría guardada información en el primer elemento de la lista de llamados REST. Segundo, los puntos habilitados después de la llamada del método que representa el `RestCallResult` actual (`enabledEndpointsAfterAction`). Junto con esta información guardamos el verbo HTTP y la ruta de la acción para facilitar la construcción de la EPA más adelante. Por último, también se guarda un valor booleano `isInitialAction`. Este nos indica si el llamado REST que tiene guardado `enabledEndpointsBeforeAction` es una acción inicial o si se inicializó la base de datos antes. Es decir, podemos saber si el primer llamado HTTP del test se está realizando desde el estado inicial de la API o no.

Cuando se está calculando la aptitud de cada `Individual` i en `AbstractRestFitness`, se construye para cada `RestCallAction` a de i un `RestCallResult` r . Esta parte del algoritmo se modificó para que antes de que se invoque a , se guarden los `enabledEndpointsBeforeAction` si es la primer operación de i . Para esto se hace un llamado HTTP a `GET::/enabledEndpoints` en este momento del algoritmo.

Se transforma el `RestActionsDto.java` retornado por la llamada HTTP a la API en un `RestActions.kt` para respetar las convenciones existentes en `EvoMaster`. Este `RestActions` se guarda en r como `enabledEndpointsBeforeAction`.

Si no se inicializó la base de datos previo a a con llamados directos a la misma, entonces también almacenamos `true` en `isInitialAction`, ya que a se ejecuta con el estado de la aplicación inicial, apenas se ejecutó `resetStateOfSUT()` del driver.

Luego, se ejecutan en orden cada a . Si la respuesta a a fue exitosa, con un estado HTTP menor a 400, llamamos a `GET::/enabledEndpoints` inmediatamente después para grabar el estado de la aplicación. Realizamos la misma transformación de antes y guardamos el `RestActions` junto con el verbo y ruta de a en el r como `Enabled.kt` en `enabledEndpointsAfterAction`.

De esta manera para cada i tenemos información sobre los endpoints habilitados antes y después de cada a . Para el primer elemento de i la información previa está en `enabledEndpointsBeforeAction`. Para los demás elementos alcanza con ver los `enabledEndpointsAfterAction` de la acción anterior.

Una vez generado el `RestCallResult`, `EvoMaster` continúa generando nuevos `Individual` o mejorando los existentes, y descartando los de menor *fitness*. Una vez que se llega al límite de tiempo de generación, la herramienta optimiza y escribe los `Individual` como tests.

A continuación, procedemos con la generación de la EPA basado en la batería de tests final generada. EvoMaster verifica si la configuración `epaCalculation` está habilitada. En el caso afirmativo, la clase `EpaWriter` se encarga de generar la EPA y guardarla. Para representar la EPA internamente en EvoMaster, se crearon las clases `EPA`, `Vertex` y `Edge`.

Para generar la EPA, iteramos por todos los i de la lista de `Individual` en `Solution` y por cada rcr `RestCallResult` de i .

Si rcr tiene `enabledEndpointsBeforeAction` es porque es el primer rcr de i . En este caso, verificamos si se inicializó i con llamados SQL realizados previos a la acción que estamos evaluando. En caso afirmativo, generamos un `Vertex` normal para representar el estado previo a la acción asociada a rcr . En caso negativo, `enabledEndpointsBeforeAction` representa un estado inicial para la API, por lo que almacenamos `true` en `isInitial` en el `Vertex` generado.

Luego, evaluamos los `enabledEndpointsAfterAction`. Si estos están presentes, creamos un `Vertex` representando el estado posterior a la acción asociada a rcr . Unimos este estado con el estado anterior mediante un `Edge` etiquetado con la acción que representa. Si no está presente, es porque a no fue exitoso o el llamado HTTP para almacenar esta información falló. En este caso, descartamos el resto de este individuo para la construcción de la EPA.

Algoritmo 2 Pseudocódigo de `writeEPA()`

Input: Solución S

Output: EPA en lenguaje `.dot` E

```

1:  $E \leftarrow \text{EPA}()$ 
2: for all  $i \in \text{INDIVIDUALS}(S)$  do
3:    $\text{previousVertex} \leftarrow \text{VERTEX}()$ 
4:    $\text{currentVertex} \leftarrow \text{VERTEX}()$ 
5:   for all  $rcr \in \text{GETRESTCALLRESULTS}(i)$  do
6:      $\text{enabledBefore} \leftarrow \text{GETENABLEDENDPOINTSBEFOREACTION}(rcr)$ 
7:     if  $\text{enabledBefore} \neq \emptyset$  then
8:        $\text{isInitial} \leftarrow \text{ISINITIAL}(i)$ 
9:        $\text{previousVertex} \leftarrow \text{CREATEORGETVERTEX}(E, \text{enabledBefore}, \text{isInitial})$ 
10:    end if
11:     $\text{enabledAfter} \leftarrow \text{GETENABLEDENDPOINTS AFTER ACTION}(rcr)$ 
12:    if  $\text{enabledAfter} \neq \emptyset$  then
13:       $\text{currentVertex} \leftarrow \text{CREATEORGETVERTEX}(E, \text{enabledAfter})$ 
14:       $\text{restAction} \leftarrow \text{GETASSOCIATEDRESTACTION}(rcr)$ 
15:       $\text{ADDDIRECTEDGE}(E, \text{previousVertex}, \text{currentVertex}, \text{restAction})$ 
16:       $\text{previousVertex} \leftarrow \text{currentVertex}$ 
17:    else
18:      break ▷ because of failed HTTP requests we are missing necessary
data to build the EPA for the rest of the individual.
19:    end if
20:  end for
21: end for
22: return  $E$ 

```

Para almacenar la EPA generada, la traducimos al lenguaje DOT [9] y la guardamos como un archivo `.dot` en la ubicación configurada por el usuario con `epaFile`. Luego,

el usuario puede transformar de manera muy simple este archivo en una imagen. Para esto utilizamos Graphviz: un software *open-source* para visualización de grafos. Este tiene varias implementaciones en web o programas instalables. La opción que se utilizó para esta tesis es la herramienta de línea de comando de Graphviz [10] para generar un *epa.png* desde un *epa.dot* escribiendo lo siguiente en la terminal:

```
dot -Tpng epa.dot -o epa.png
```

3.2. Nueva heurística para EPAs en EvoMaster

Para que EvoMaster priorice los casos de test que ejercitan nuevos caminos de la EPA, agregamos al algoritmo evolutivo una nueva heurística. Esta funciona de manera muy similar a la heurística previamente existente en EvoMaster que favorece los *Individual* que encuentran nuevos códigos HTTP en las respuestas de cada endpoint de la API.

Veamos a continuación la implementación. Primero se creó una configuración *heuristicsForEpa* para poder activar o desactivar este heurística. Esta requiere que esté activada *epaCalculation*.

Luego se modificó cómo se calcula el *fitness* de un *Individual*. Para cada *RestCallResult rcr*, si fue un llamado exitoso a la API, se incluye un *target* con la arista de la EPA que representa *rcr*. Identificamos esta heurística como *vértice:arco:vértice* o *habilitadosAntesDeAcción:acción:habilitadosDespuésDeAcción*. El pseudocódigo de este mecanismo se ve en el algoritmo 3.

De esta manera, si un *Individual* “descubre” muchos arcos de la EPA, cubrirá muchos *targets*, y MIO lo priorizará de manera acorde.

3.3. Discusión

Como cierre a esta sección, es importante notar que tanto la generación de EPAs como la nueva heurística tienen un costo computacional alto, lo cual puede tener un impacto negativo en las test suites generadas. Esto se debe a que estas extensiones a EvoMaster impactan en el tiempo de ejecución de cada búsqueda del algoritmo MIO. En particular, los llamados HTTP a *GET::/enabledEndpoints* entre cada acción de cada test pueden ser computacionalmente muy costosos.

Si el tiempo de ejecución de cada búsqueda de MIO es mayor, se llevarán a cabo menos iteraciones del algoritmo cuando la generación de EPAs esté habilitada, en comparación con una ejecución sin la generación de EPAs que tenga la misma duración. Esto impactará directamente en la cantidad de nuevos tests aleatorios y mutaciones que analizará MIO, y puede afectar cualitativa o cuantitativamente las test suites generadas por EvoMaster.

Algoritmo 3 Pseudocódigo de `handleEpaTargets()`

Input: Valor de la función de fitness del test fv , Lista de resultados de las acciones del test R

```

1:  $actionIndex \leftarrow 0$ 
2: for all  $r \in R$  do
3:   if GETSTATUSCODE( $r$ )  $\geq 400$  then
4:     return  $\triangleright$  si el llamado HTTP no fue exitoso entonces no tendremos
               nueva cobertura de la EPA
5:   end if
6:    $actionName \leftarrow \text{GETACTIONNAME}(r)$ 
7:    $previousEnabled \leftarrow \text{GETENABLEDENDPOINTSBEFOREACTION}(r)$ 
8:   if  $previousEnabled = \emptyset$  and  $actionIndex > 0$  then
9:      $\triangleright$  si  $r$  no es la primera acción del test, obtenemos los  $previousEnabled$ 
        de  $r$  de los  $EnabledEndpointsAfterAction$  de la acción anterior.
10:     $prevR \leftarrow \text{GETRESULT}(actionIndex - 1, R)$ 
11:     $previousEnabled \leftarrow \text{GETENABLEDENDPOINTS AFTER ACTION}(prevR)$ 
12:  end if
13:   $currentEnabled \leftarrow \text{GETENABLEDENDPOINTS AFTER ACTION}(r)$ 
14:  if  $previousEnabled \neq \emptyset$  and  $currentEnabled \neq \emptyset$  then
15:     $edgeId \leftarrow currentEnabled : actionName : previousEnabled$ 
16:     $edgeTarget \leftarrow \text{HANDLELOCALTARGET}(targetName)$ 
17:     $\triangleright$  actualizamos el target con valor 1 que representa cubierto
18:     $\text{UPDATETARGET}(fv, edgeTarget, 1.0, actionIndex)$ 
19:  end if
20:   $actionIndex \leftarrow actionIndex + 1$ 
21: end for
22: return  $E$ 

```

4. EXPERIMENTACIÓN

Para la experimentación usaremos tres REST APIs: **employees**, **features-service** y **proxyprint**. La primera es la misma que se presentó en la sección 3. Las dos últimas son parte de EMB (EvoMaster Benchmark) [11], una compilación de APIs para experimentar con EvoMaster.

Para cada una de las APIs analizaremos y compararemos resultados con y sin las heurísticas para EPAs. El objetivo es medir el impacto y efectividad de la nueva heurística.

Ya que EvoMaster no es determinístico, se decidió correr cada variante de la experimentación diez veces. Además, para EvoMaster en general se recomienda limitar el tiempo entre 1 y 24 horas para obtener los mejores resultados para baterías de test. Es por esto que se decidió limitar cada una de las búsquedas en una hora. En total esta experimentación tomó $3 \text{ APIs} \times 2 \text{ casos} \times 10 \text{ repeticiones} \times 1 \text{ hora} = 60\text{hs}$ de cómputo.

Luego, se ejecutó EvoMaster con y sin heurísticas para EPAs con las APIs **features-service** y **proxyprint** limitando la búsqueda en 1, 15 y 30 minutos y 1, 3 y 6 horas. En este caso, cada variante se ejecutó tres veces. Esta experimentación tomó $2 \text{ APIs} \times 2 \text{ casos} \times 3 \text{ repeticiones} \times (1\text{m}+15\text{m}+30\text{m}+1\text{h}+3\text{h}+6\text{h}) \approx 130\text{hs}$ de cómputo.

Para todas estas corridas se almacenó la cantidad de aristas de la EPA generada.

Finalmente, nuestro objetivo es evaluar el impacto de las extensiones implementadas en EvoMaster en las test suites. Para ello, se llevó a cabo la ejecución de EvoMaster en tres configuraciones distintas: sin generación de EPAs, con generación de EPAs sin heurística, y con generación de EPAs con heurística para la API **features-service**. Esta experimentación se realizó diez veces para cada variante con duración de media hora cada una. Esto tomó $3 \text{ casos} \times 10 \text{ repeticiones} \times 30 \text{ minutos} = 15\text{hs}$ de cómputo. En este caso se almacenó para cada ejecución el número de tests generados, targets cubiertos y acciones y tests evaluados.

La experimentación se realizó en una computadora con un procesador Apple M1 Max con 32GB de RAM y MacOS Sonoma 14.3.1.

Algo a resaltar es que para toda la experimentación, se configuró `generateSqlDataWithSearch` y `generateSqlDataWithDSE` en `false`, para evitar que EvoMaster modifique directamente la base de datos. De esta manera la EPA no tendrá estados desconectados, y todos sus estados tienen al menos un camino desde el estado inicial.

4.1. REST APIs utilizadas en la experimentación

4.1.1. Employees

Como se presentó anteriormente, **employees** es una REST API CRUD con registros del tipo **Employee**. Fue generada específicamente para esta tesis para la evaluar la generación de EPAs con EvoMaster en una API simple. Sus endpoints se pueden observar en la figura 3.1. Es muy intuitivo determinar cuando está habilitado cada endpoint, y la consecuente implementación de `GET::/enabledEndpoints` (extracto de código 3.1).

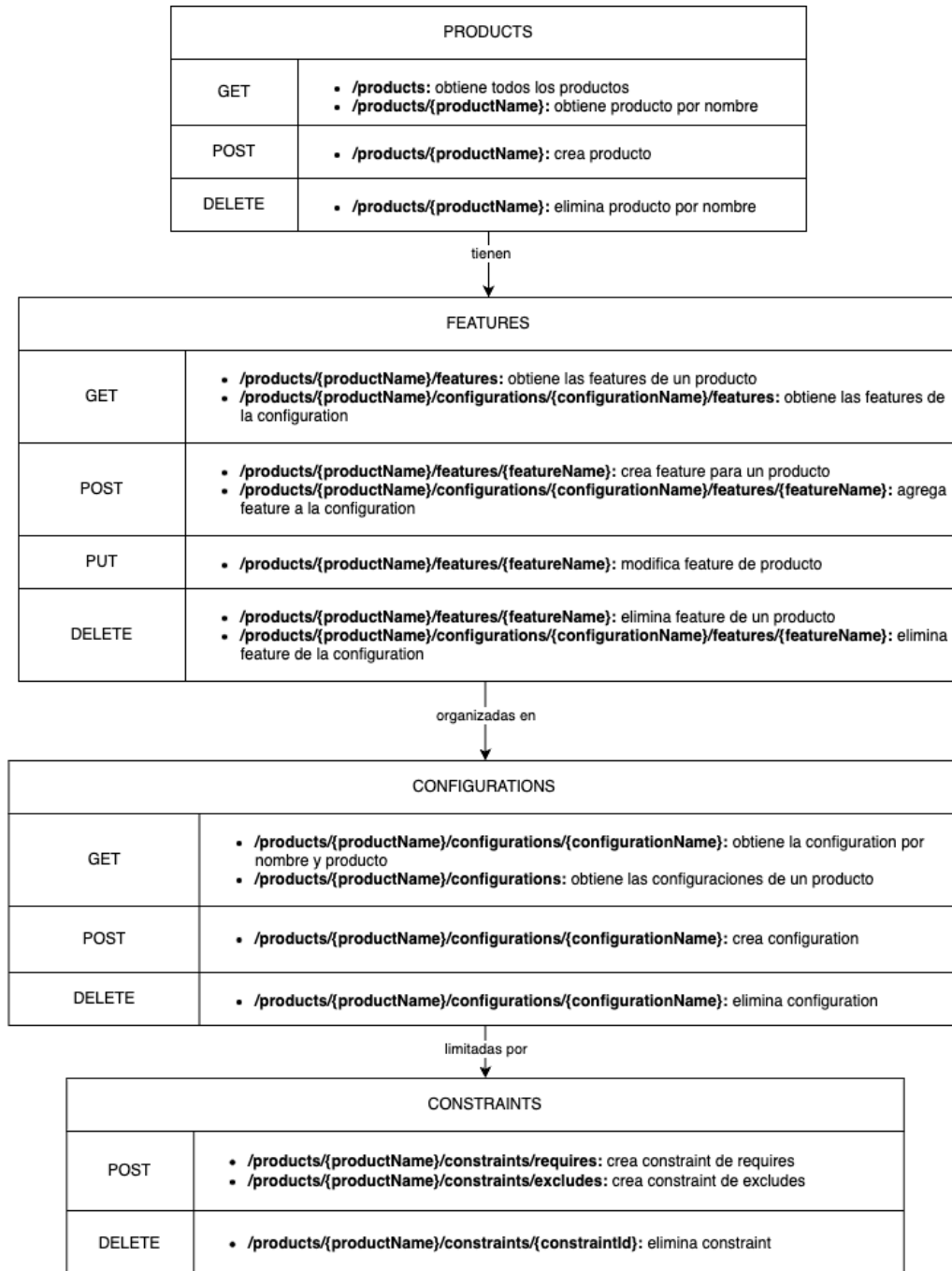


Fig. 4.1: Endpoints de features-service.

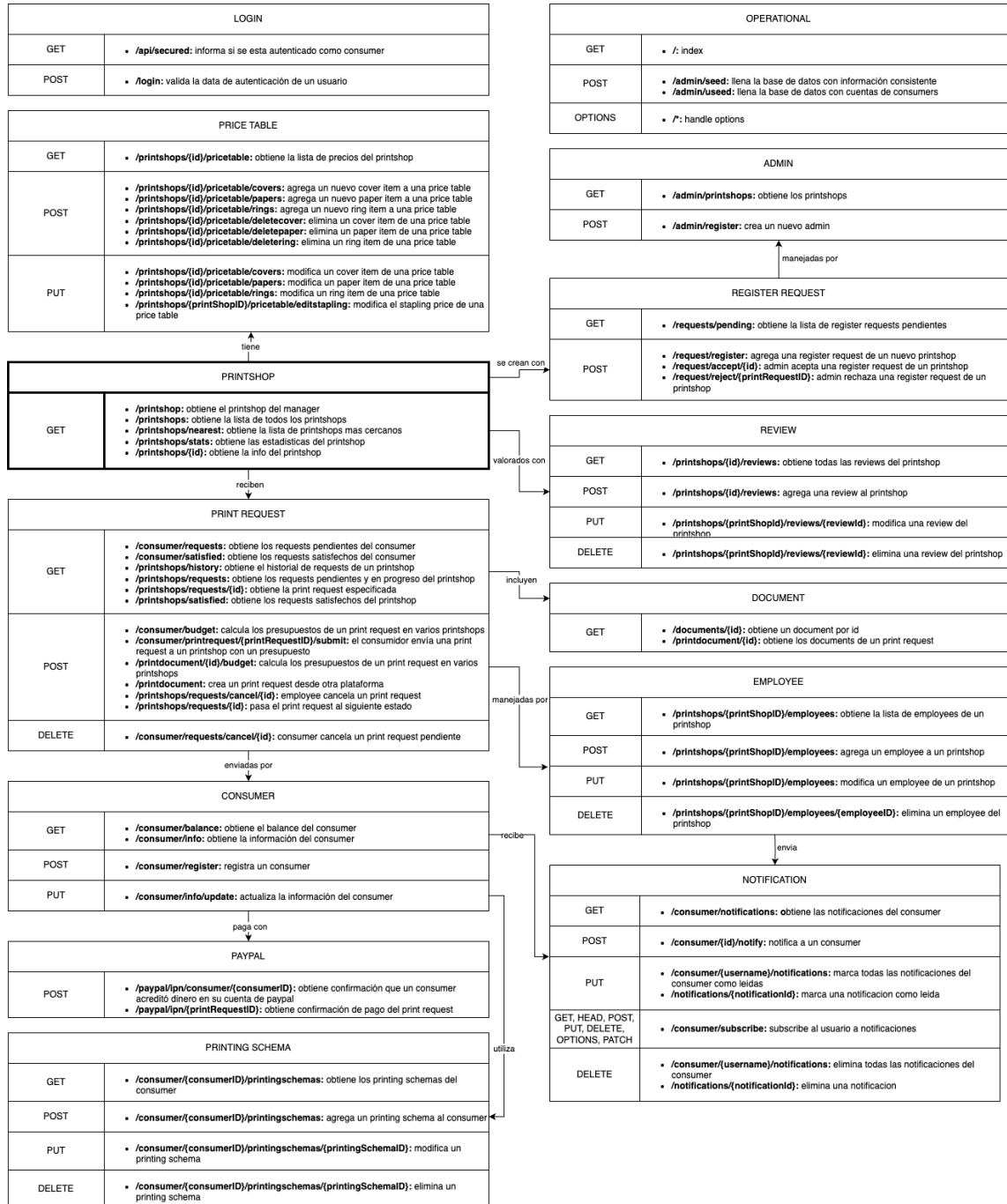


Fig. 4.2: Endpoints de proxyprint.

```

8      "/beans", "/beans.json",
9      "/configprops", "/configprops.json",
10     "/dump", "/dump.json",
11     "/env", "/env.json", "/env/{name}",
12     "/error",
13     "/health", "/health.json",
14     "/info", "/info.json",
15     "/mappings", "/mappings.json",
16     "/metrics", "/metrics.json", "/metrics/{name}",
17     "/trace", "/trace.json",
18     "/enabledEndpoints")
19 );
20 }

```

Extracto de código 4.1: `getProblemInfo()` de `proxyprint`.

En la figura 4.2 podemos ver los endpoints de `proxyprint` que se analizaron con EvoMaster. Podemos ver que hay una gran cantidad de endpoints en esta REST API, pero esto no necesariamente esto se traducirá en una gran cantidad de estados en la EPA.

Pensando en cómo implementar `GET::/enabledEndpoints` podemos definir las siguientes categorías de configuraciones:

- Sin nada: no nos encontraremos este caso ya que inicializamos la base de datos antes de cada test y no hay métodos para eliminar *printshops*.
- Con *printshops*: habilita todos los métodos para operar sobre los mismos.
- Con *registered requests*: habilita aceptar y rechazarlas.
- Con consumidores: permite que los consumidores operen.
- Con notificaciones: permite modificar y eliminar notificaciones existentes.
- Con *print requests*: permite enviar, pagar, obtener y eliminar *print requests*.
- Con *printing schemas*: permite operar sobre los mismos.
- Con *managers/empleados*: habilita acceder a información sobre el *printshop*, los empleados y las tareas pendientes.
- Con reseñas: permite modificar y borrarlas.
- Con *cover items*: permite modificarlas.

De igual modo que en la API anterior, estas categorías de configuraciones son combinables entre ellas.

Por último, es importante destacar que esta es la única API de la experimentación para la cual hay endpoints que requieren autenticación. En este caso hay los roles `admin`, `consumer`, `manager` y `employee`.

Esto puede tener un impacto en la cobertura lograda por EvoMaster de la EPA ya que a la dificultad de encontrar combinaciones de parámetros y llamados HTTP exitosos, se le suma el requerimiento de que la mayoría de estos llamados necesitan credenciales de autenticación específicas. La única manera que tiene EvoMaster de saber cuál es la combinación correcta de endpoint y autenticación es con prueba y error.

4.2. Resultados

En esta sección se presentan los resultados de la experimentación de generación de subaproximaciones de EPAs con EvoMaster.

Para esto, realizamos un análisis comparativo entre dos variantes \mathcal{H} y \mathcal{NH} , las cuales representan la generación de EPAs para una API con y sin la aplicación de las nuevas heurísticas. El objetivo es medir el impacto de las heurísticas en la eficiencia y el rendimiento de la generación de EPAs más completas.

Para llevar a cabo esta evaluación, aplicamos el test estadístico no paramétrico *U de Mann-Whitney*, una herramienta para comparar dos muestras independientes y determinar si provienen de la misma población. Nuestra hipótesis nula postula que no existe diferencia estadísticamente significativa entre los resultados obtenidos al aplicar las heurísticas para EPA y aquellos obtenidos sin su implementación.

Para el test de hipótesis nula usaremos el *p-valor*: la probabilidad de obtener resultados al menos tan extremos como los observados, asumiendo que la hipótesis nula es correcta. Un *p-valor* pequeño nos dice que el escenario observado es muy poco probable si la hipótesis nula es verdadera. Determinamos que la hipótesis nula queda descartada si $p \leq 0,05$.

En el caso de que las variantes \mathcal{H} y \mathcal{NH} muestren una diferencia estadística, utilizamos la medida \hat{A}_{12} de Vargha y Delaney [15] para cuantificarla. La fórmula de esta medida es:

$$\hat{A}_{12} = \frac{R_1/m - (m+1)/2}{n}$$

con:

- m y n la cantidad de observaciones de la primera y segunda muestra de datos, respectivamente.
- R_1 la suma del ranking de la primera muestra de datos:

$$R_1 = \frac{m(m+1)}{2} + \#(X_i > Y_j) + 0,5\#(X_i = Y_j)$$

Esta medida se interpreta de manera sencilla. Supongamos que tenemos dos variantes \mathcal{A} y \mathcal{B} , $\hat{A}_{12} = 0,55$ significa que \mathcal{A} obtiene mejores resultados el 55 % de los experimentos. Diversos autores caracterizan estos valores con distintos límites, pero en este caso usaremos las definiciones de Vargha y Delaney. Ellos postulan que $\hat{A}_{12} \geq 0,56$ es una diferencia pequeña, $\hat{A}_{12} \geq 0,64$ es mediana y $\hat{A}_{12} \geq 0,71$ es grande.

En la tabla 4.1 se exhiben los *p-valores* obtenidos con el test *U de Mann-Whitney* y las medidas \hat{A}_{12} de Vargha y Delaney.

Las figuras 4.3, 4.5 y 4.8 muestran los diagramas de caja que ilustran los resultados de la experimentación en cuanto a cantidad de aristas encontradas en todas las ejecuciones.

Basado en los *p-valores*, podemos decir que la heurística para EPAs tuvo un impacto estadísticamente significativo en el número de aristas encontradas para **employees** y **features-service**. En cambio, para **proxyprint** no fue estadísticamente significativo.

Observando los resultados de la \hat{A}_{12} de Vargha y Delaney, podemos decir que la diferencia estadística entre los resultados con y sin heurística fue grande para **employees** y **features-service**. Es interesante notar que para **proxyprint**, en esta medida la diferencia estadística es pequeña.

Podemos decir que se encontraron resultados mejores en cuanto a cobertura de aristas encontrados en las EPAs de todas las APIs en el 60 % o más de las veces con heurísticas

REST API	employees	features-service	proxyprint
p -valor	1.59379e-05	0.00018	0.46864
\hat{A}_{12} de Vargha y Delaney	1	1	0.6

Tab. 4.1: p -valores obtenidos del test U de Mann-Whitney y \hat{A}_{12} de Vargha y Delaney para la cantidad de aristas de las EPAs obtenidas en ejecuciones sin vs. con heurísticas para EPAs.

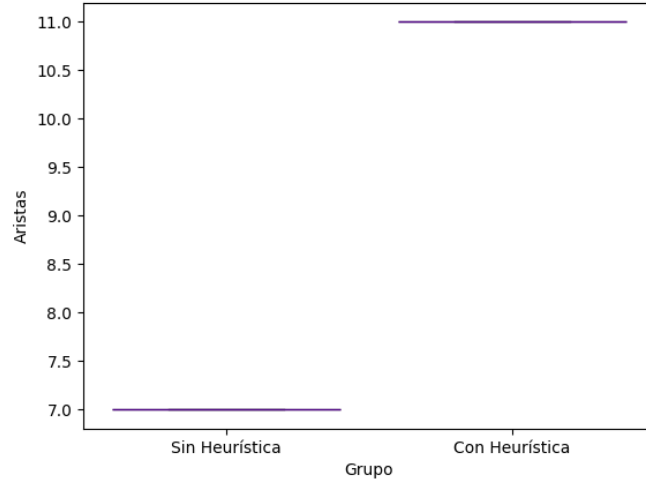


Fig. 4.3: Aristas para **employees** sin vs. con la nueva heurística.

para EPAs. Para **employees** y **features-service**, los resultados con heurísticas fueron mejores en el 100 % de los casos.

En la figura 4.3 podemos ver claramente el impacto de la heurística en **employees**. El promedio de la cantidad de aristas es mucho mayor para las EPAs generadas con la misma.

Es interesante resaltar que para **employees** no solo la cantidad de aristas encontradas es la misma en las diez ejecuciones de la experimentación con y sin heurísticas respectivamente (4.3), si no que las EPAs generadas son exactamente iguales para cada variante.

En la figura 4.4 podemos los dos estados de esta API: con y sin empleados. Como la “base de datos” esta vacía en la inicialización, el estado inicial es sin empleados. Al agregar un empleado cambiamos de estado, mientras que al remover todos los empleados volvemos al estado inicial. Los demás métodos de la API nos retornan al mismo estado que estábamos antes de ser llamado.

Al ser una API muy simple, en poco tiempo y con la heurística implementada se generó la EPA completa. La única arista que se podría argumentar que falta es `/v2/api-docs::GET` en el segundo estado, pero este no es un endpoint que aporte al comportamiento de la API. Es el que informa al usuario sobre la estructura de la API, no tiene ningún efecto sobre el estado de la misma. Pero podemos ver que para ambos nodos se llamó a todos los métodos que tienen habilitados en cada uno.

Para la EPA sin heurísticas es interesante que no haya encontrado más aristas en ninguna de las ejecuciones. Probablemente esto se deba a que con las heurísticas preexistentes

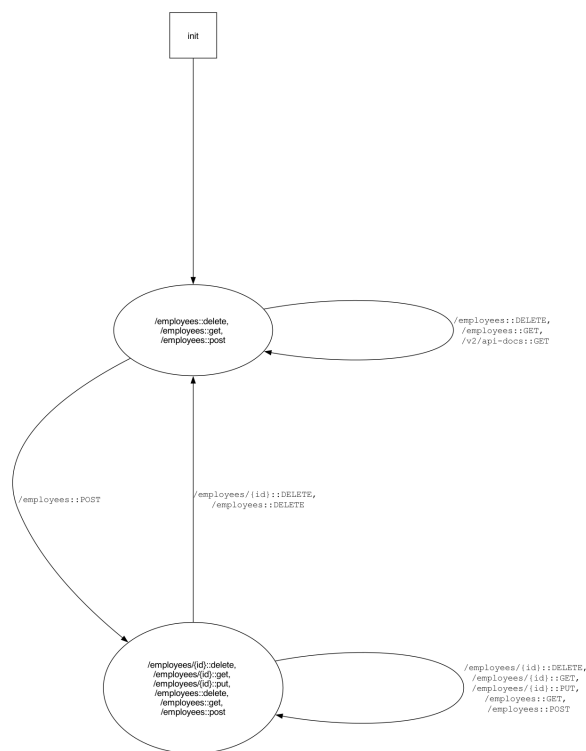


Fig. 4.4: EPA generada para `employees` con heurísticas en 1 hora.

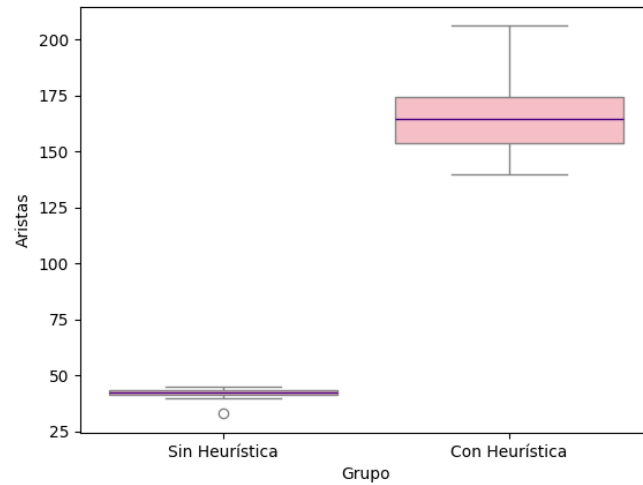


Fig. 4.5: Aristas para `features-service` sin vs. con la nueva heurística.

no hay targets específicos por los que se guarden los casos de test que completan la EPA.

En `features-service` también impactó mucho la nueva heurística en la cantidad de aristas de la EPA encontradas. En la figura 4.5 podemos observar que todas las EPAs obtenidas mediante la heurística presentan un mayor número de aristas en comparación con aquellas generadas sin ella.

También es la API que generó las EPAs más complejas. Como ejemplo, se puede observar en la figura 4.6 la EPA con más aristas encontradas en una hora en la experimentación.

Algo a resaltar es que en todas las EPAs generadas de `features-service` podemos encontrar errores en la implementación de la API. Por ejemplo, en la figura 4.7 se observa que en el estado inicial se supone que solo deberían estar habilitados los métodos `/products/{productName}::POST` y `/products::GET` pero EvoMaster pudo llamar exitosamente a

- `/products/{productName}/configurations/{configurationName}::GET`,
- `/products/{productName}/configurations::GET`, y
- `/products/{productName}/constraints/{constraintId}::DELETE`

desde este estado. Esto ejemplifica la utilidad de la subaproximación de la EPA como complemento a los tests de EvoMaster para encontrar errores de implementación.

Es importante recordar que `features-service` es una APIs parte de EMB, una compilación de APIs para *benchmarking* de EvoMaster que contienen errores para detectar.

Por último, algo a destacar de las experimentaciones de `proxyprint` es que en la figura 4.8 vemos claramente como los resultados de las experimentaciones no son muy distintos entre si, pero con la heurística son un poco mejores en promedio. Se puede observar que en algunos casos sin heurística se obtuvieron mejores resultados, esto puede atribuirse a la aleatoriedad de las mutaciones de MIO.

En la figura 4.9 se observa la EPA más completa obtenida en la experimentación en una hora. Se puede resaltar como en el caso de esta API se encuentra un gran número de aristas pero sólo dos estados. Vemos que se llega al segundo estado después de llamar

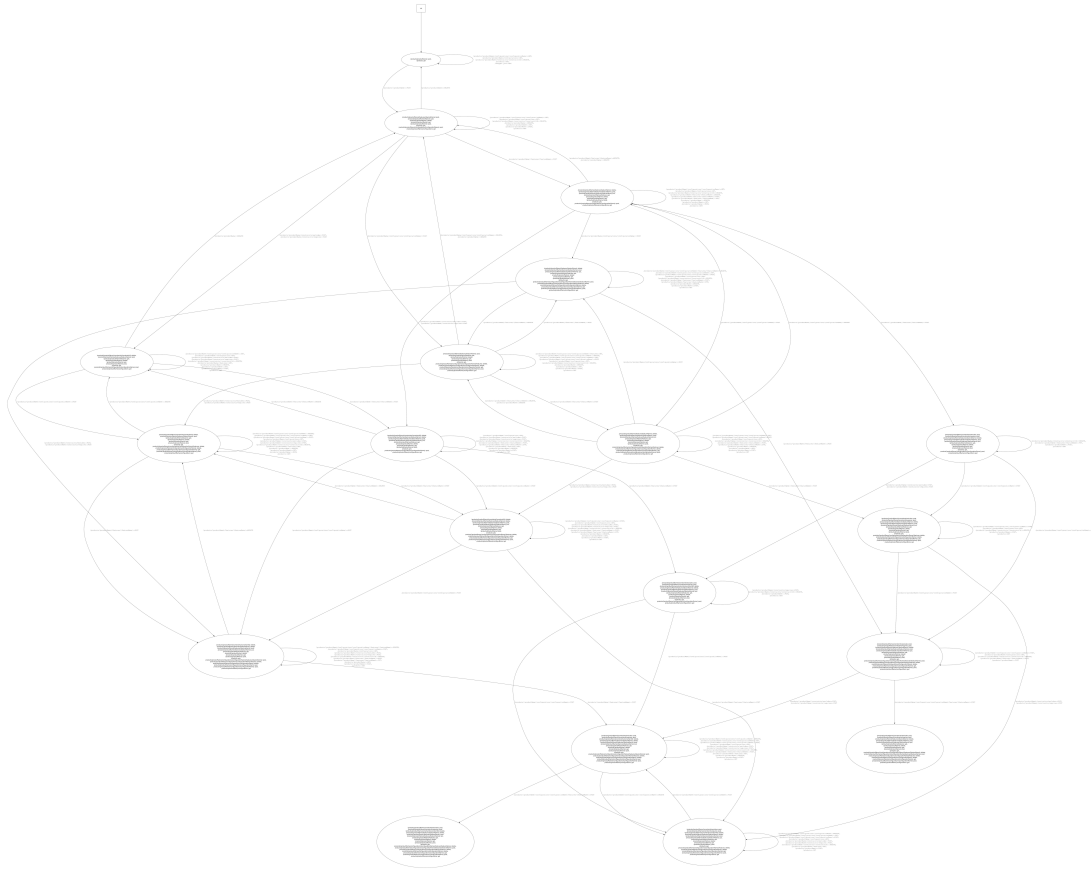


Fig. 4.6: EPA generada para **features-service** con heurísticas en 1 hora.

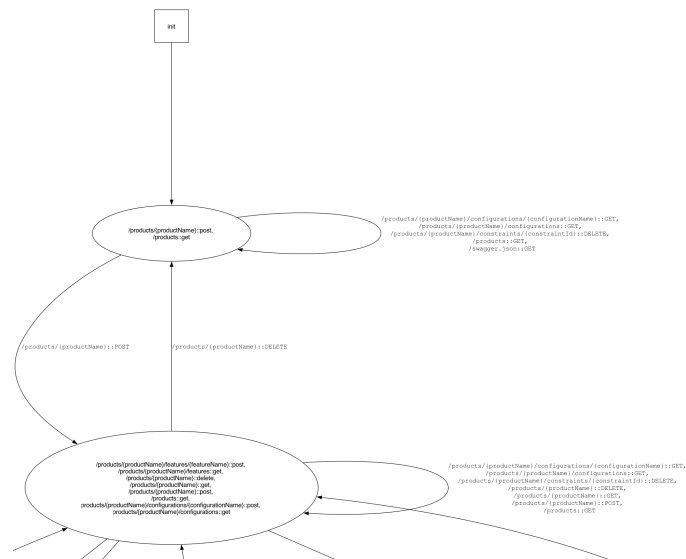


Fig. 4.7: Zoom de la figura 4.6.

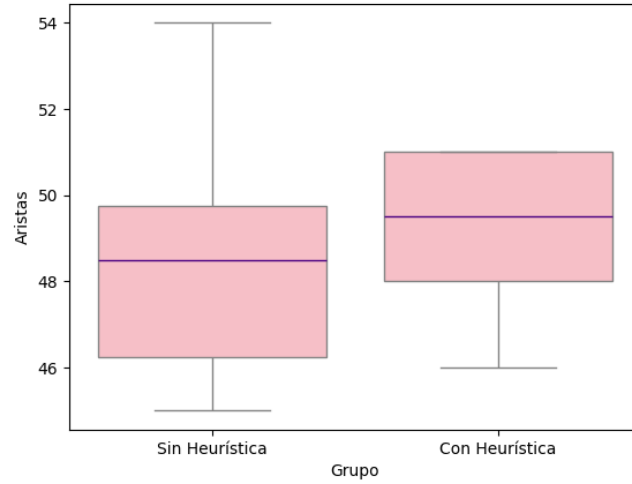


Fig. 4.8: Aristas para `proxyprint` sin vs. con la nueva heurística.

sin vs. con generación de EPAs con heurísticas				
Medida	Tests generados	Targets cubiertos	Acciones evaluadas	Tests evaluados
$p\text{-valor}$	0.00017	0.00018	0.00018	0.00018
\hat{A}_{12}	0	0	1	1

sin vs. con generación de EPAs sin heurísticas				
Medida	Tests generados	Targets cubiertos	Acciones evaluadas	Tests evaluados
$p\text{-valor}$	0.00017	0.00018	0.00018	0.00018
\hat{A}_{12}	1	0	1	1

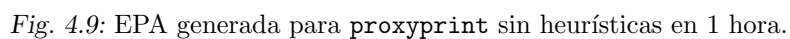
Tab. 4.2: $p\text{-valores}$ obtenidos del test U de Mann-Whitney y \hat{A}_{12} de Vargha y Delaney sobre las test suites de EvoMaster.

a `/consumer/id/notify::POST`. Al tener notificaciones en esa configuración, los endpoints `/notifications/notificationId::PUT` y `/notifications/notificationId::DELETE` están habilitados en el segundo estado.

Hay solo un endpoint de la API que no está habilitado en ninguno de los estados encontrados: `/documents/id::GET`. Esto puede ser porque EvoMaster no lo encontró en el tiempo dado o algún error de la implementación no permite que este método pueda ser llamado exitosamente con ninguna configuración. Pero esto sugiere que teóricamente podría haber al menos un estado más en la EPA.

En la figura 4.10 podemos ver cómo varió la cantidad de arcos encontrados para `features-service` y `proxyprint` con el tiempo. Podemos ver que el impacto positivo de la heurística en la cantidad de aristas encontradas se mantiene ejecutando EvoMaster con distintos límites de tiempo para estas APIs.

Podemos observar como para ambas APIs, pero especialmente para `features-service`, la diferencia en la cantidad de aristas encontradas con las dos variantes creció a mayor tiempo de ejecución. También podemos afirmar que ejecuciones más prolongadas de EvoMaster sin la heurística obtienen EPAs más completas.



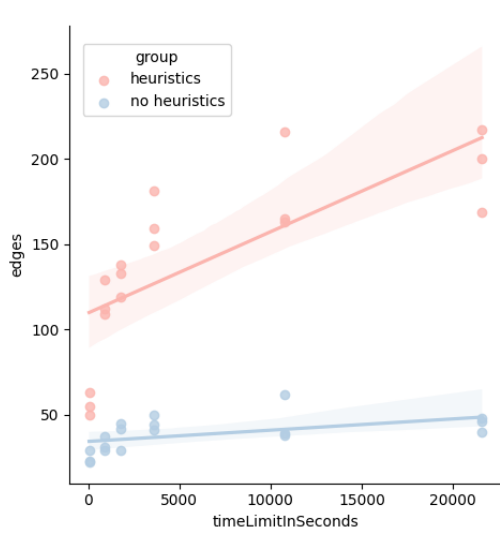
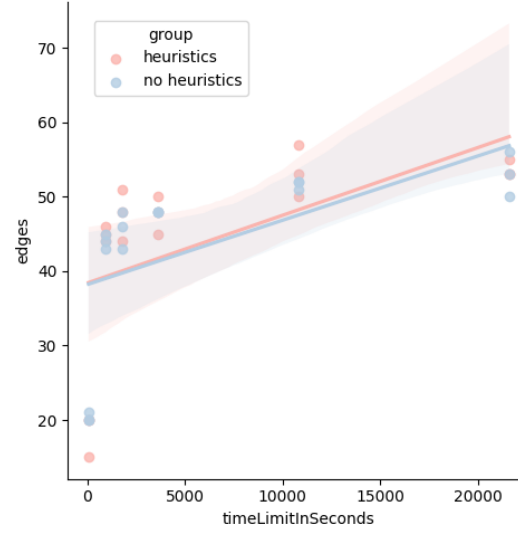
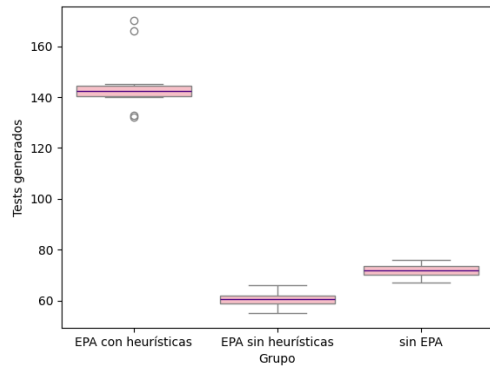
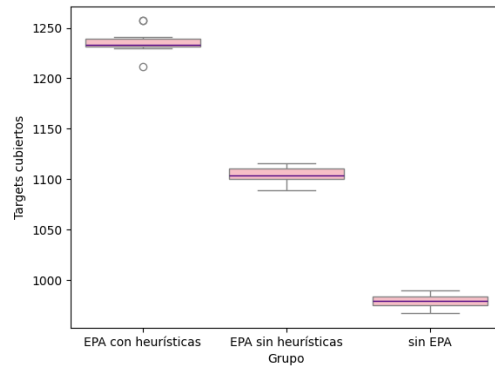
(a) Para **features-service**.(b) Para **proxyprint**.

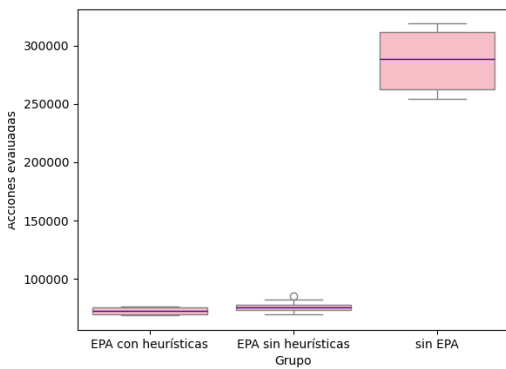
Fig. 4.10: Gráficos de dispersión de la cantidad de arcos encontrados en 1m, 15m, 30m, 1h, 3h y 6h.



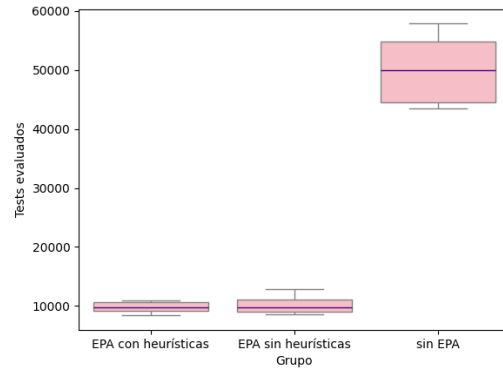
(a) Tests generados.



(b) Targets cubiertos.



(c) Acciones evaluadas.



(d) Tests evaluados.

Fig. 4.11: Valores de test suites de EvoMaster.

Por último, para analizar el impacto de la generación de EPAs en las test suites generadas utilizaremos el mismo análisis comparativo realizado previamente con la *U de Mann-Whitney* y \hat{A}_{12} de *Vargha y Delaney*. Podemos observar en la tabla 4.2 los valores obtenidos. El algoritmo MIO tiene como objetivo generar una test suite que cubra la mayor cantidad de targets con la menor cantidad de tests cortos. Estas son las características con las que determinaremos la calidad de una test suite en esta sección.

Basado en los p-valores, podemos decir que generar EPAs tuvo un impacto estadísticamente significativo en la cantidad de tests generados, targets cubiertos y acciones y tests evaluados.

Observando los resultados de la \hat{A}_{12} de *Vargha y Delaney*, podemos decir que la diferencia estadística fue grande para todas las variantes. En los casos en que la medida es cero es porque se obtuvieron mayores valores con la generación de EPAs como se puede observar en la figura 4.11. En otras palabras, en estos casos se obtienen mejores resultados sin generar la EPA el 0 % de los experimentos.

Observando la figura 4.11, comparando los resultados obtenidos generando la EPA con heurísticas puede surgir la pregunta: “¿Cómo se cubren más targets y se generan más tests habiendo evaluado muchas menos acciones y tests?”. Esto se debe a que al incluir las heurísticas para EPAs hay nuevos targets, uno para cada arista. La cantidad de targets está directamente relacionada con la cantidad de tests que se evalúan con el algoritmo MIO, como se explicó en la sección 3. De todos modos, puede que esto no explique todos los nuevos targets encontrados.

Por esta razón, evaluamos el impacto de la generación de EPAs en la suite de tests considerando los resultados para EPAs sin heurísticas como referencia. De esta manera establecemos una cantidad más similar de targets y que permita una comparación más precisa.

El hecho de estar ejecutando el código de `GET::/enabledEndpoints` en los casos que generamos EPAs también puede estar afectando los targets cubiertos por los tests. Esto se debe a que la cobertura de líneas de código y *branches* son targets de EvoMaster, y pueden estar registrándose como de targets cubiertos en estos casos.

Es importante destacar que, en contraste con los resultados obtenidos sin generar EPAs, generando EPAs sin heurísticas se cubren más targets con menos tests, habiendo evaluando muchas menos acciones y tests. A pesar de ser contraintuitivo, esta es una tendencia consistente en los datos que merece una mayor investigación y análisis para comprender completamente su origen. De todos modos, las ejecuciones de EvoMaster generando EPAs sin heurísticas parecen generar las mejores test suites entre los casos analizados.

5. TRABAJO FUTURO

A continuación se presentarán diversas oportunidades para extender y mejorar este trabajo en el futuro:

- Implementación de EPAs en EvoMaster para otros tipos de APIs: sería interesante poder generar EPAs para APIs GraphQL [16] y RPC [17].
- Otras heurísticas: Incluyendo nuevas heurísticas se podría generar EPAs más eficientemente. Por ejemplo, una heurística que favorezca los casos de test que encuentran nuevos nodos de la EPA.
- Análisis de endpoints habilitados: Se podría utilizar la implementación de `GET::/enabledEndpoints` para verificar que los endpoints cuyos llamados son exitosos estaban efectivamente habilitados previo a su llamado y viceversa. Esta información se puede compilar como un reporte, archivo de texto o un test que debería fallar.
- Computar qué métodos están habilitados automáticamente: Sería interesante poder generar la EPA sin la necesidad de implementar `/enabledEndpoints::GET`.
- Evaluar por qué generando EPAs sin heurísticas se obtienen mejores test suites: En la experimentación fue el caso en el que se cubren más targets con menos tests, habiendo evaluando muchas menos acciones y tests. Saber por que pasa esto puede derivar en mejoras para EvoMaster.

6. CONCLUSIONES

Este trabajo presenta una innovadora metodología para generar subaproximaciones de EPAs para REST APIs, implementada como una nueva funcionalidad en EvoMaster, una herramienta de generación de tests automatizados con algoritmos evolutivos.

A diferencia de los métodos estáticos previos, esta propuesta permite una generación dinámica de la EPA. Para potenciar aún más la completitud de las EPAs generadas, se implementó una heurística en EvoMaster que prioriza los casos de test que descubren nuevas aristas en la EPA.

Una evaluación experimental con diversas APIs demostró el impacto estadísticamente significativo de la heurística en la cantidad de aristas encontradas, mejorando considerablemente la cobertura de las EPAs.

En conclusión, las funcionalidades desarrolladas en esta tesis son una valiosa contribución a EvoMaster, potenciando la capacidad de los usuarios para validar y verificar sus REST APIs. La abstracción de estados y sus transiciones no solo facilita la comprensión del funcionamiento del programa, sino que también permite visualizar la cobertura de tests realizada por EvoMaster. El usuario puede ver fácilmente si hubo algún caso de uso de la aplicación que EvoMaster no encontró. A pesar del alto costo computacional, en los casos de uso correctos la generación de EPAs puede aportar gran valor a la validación y verificación de REST APIs.

Las EPAs generadas con la heurística implementada logran un alto nivel de completitud en poco tiempo de generación, incluso para APIs complejas. Esto se traduce en un beneficio tangible para los usuarios de EvoMaster.

Las contribuciones de esta tesis a EvoMaster son de gran utilidad y funcionan correctamente. La construcción automatizada de EPAs representa un gran avance para la herramienta y tiene el potencial de mejorar significativamente la experiencia de los usuarios en el proceso de validación y verificación de REST APIs.

Bibliografía

- [1] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. *University of California, Irvine*, 2000.
- [2] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology*, 28(1):1–37, January 2019.
- [3] Andrea Arcuri, Man Zhang, Asma Belhadi, Juan Pablo Galeotti, , Bogdan, Seran, Amid Golmohammadi, Alberto Martín López, Hghianni, Onur D, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. Emresearch/evomaster: v2.0.0, 2023.
- [4] Andrea Arcuri. *Many Independent Objective (MIO) Algorithm for Test Suite Generation*, page 3–17. Springer International Publishing, 2017.
- [5] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for rest apis: no time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA ’22. ACM, July 2022.
- [6] Andrea Arcuri. Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technology*, 104:195–206, 2018.
- [7] Guido De Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodology*, 22(3):1–46, July 2013.
- [8] EMResearch. EvoMaster Driver. https://github.com/EMResearch/EvoMaster/blob/master/docs/write_driver.md. [Online].
- [9] Graphviz. DOT Language. <https://graphviz.org/doc/info/lang.html>, 2024. [Online].
- [10] Graphviz. Command Line. <https://graphviz.org/doc/info/command.html>, 2024. [Online].
- [11] Andrea Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan Pablo Galeotti, and Seran. Emb: A curated corpus of web/enterprise applications and library support for software testing research., 2023.
- [12] JavierMF. features-service. <https://github.com/JavierMF/features-service>, 2024. [Online].
- [13] Wikipedia. Feature Model. https://en.wikipedia.org/wiki/Feature_model, 2024. [Online].
- [14] ProxyPrint. proxyprint-kitchen. <https://github.com/ProxyPrint/proxyprint-kitchen>, 2024. [Online].

-
- [15] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
 - [16] GraphQL. <https://graphql.org/>, 2024. [Online].
 - [17] Bruce Jay Nelson. *Remote procedure call*. Carnegie Mellon University, 1981.