Inferencia Automática de Dependencias Inter-Parámetro en APIs REST

A. Giuliano Mirabella, Alberto Martin-Lopez,
Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés

SCORE Lab, I3US Institute, Universidad de Sevilla, Seville, Spain
{amirabella,alberto.martin,sergiosegura,lvalencia,aruiz}@us.es

Resumen La generación automática de casos de prueba para APIs REST es un tema de investigación muy activo. La mayoría de técnicas emplean un enfoque de caja negra basado en la generación aleatoria de peticiones a partir de la especificación de la API. Dichas técnicas tienen una limitación importante: ignoran las dependencias entre parámetros. Como resultado, la mayoría de peticiones viola alguna dependencia y son rechazadas por la API. En este artículo, proponemos inferir automáticamente dichas dependencias únicamente a partir de la especificación de la API y sus entradas y salidas. Nuestra técnica aprende a medida que genera casos de prueba, de forma que el porcentaje de llamadas válidas aumenta progresivamente hasta alcanzar una precisión del 90 % en APIs comerciales como GitHub o YouTube. Estos prometedores resultados sugieren que nuestra propuesta podría mejorar significativamente la generación automática de casos de prueba para APIs REST.

Palabras clave: APIs RESTful · servicios web · pruebas software.

1. Introducción

Las APIs web permiten la interacción de sistemas software heterogéneos a través de Internet [8]. Desempeñan un papel fundamental en la integración de software, lo que se refleja en el tamaño de repositorios populares de APIs web tales como ProgrammableWeb [16], que actualmente indexa más de 24K APIs. Las APIs web modernas suelen seguir los principios de la arquitectura REST [5], recibiendo el nombre de *APIs RESTful* (o simplemente *APIs REST*). Las APIs REST proporcionan una interfaz uniforme para acceder y manejar recursos (p. ej., un vídeo en la API de YouTube) mediante peticiones HTTP. Este tipo de APIs suelen describirse con lenguajes de especificación como OpenAPI Specification (OAS) [15], considerado el estándar en la industria. OAS está diseñado para proporcionar una descripción estructurada y *machine-readable* de una API REST, lo que permite automatizar multitud de tareas como la generación de código (clientes y servidores) o las pruebas.

Las APIs REST suelen imponer restricciones entre los parámetros de sus operaciones. Por ejemplo, en la operación de búsqueda de la API de YouTube [20], al establecer el valor del parámetro `videoDefinition` a `'high'` (para buscar

vídeos en alta definición), el parámetro `type` debe ser igual a `'video'`, de otro modo la llamada será inválida y la API devolverá un código 400 ("error de cliente"). Este tipo de restricciones se denominan *dependencias inter-parámetro* (o simplemente *dependencias*), y son extremadamente comunes en las APIs REST, estando presentes en 4 de cada 5 APIs industriales de acuerdo a un estudio reciente [11]. A pesar de ello, no están soportadas por lenguajes de especificación como OAS,[1] por lo que no es posible gestionarlas de manera automática.

Debido al papel clave de las APIs REST en la integración de aplicaciones, la automatización de pruebas en este dominio se ha convertido recientemente en un tema de investigación muy activo [2,3,4,9,17,19]. Sin embargo, las técnicas actuales, basadas en OAS, ignoran las dependencias inter-parámetro, por lo que su efectividad es limitada para APIs donde abunden estas dependencias. Por ejemplo, en un estudio reciente observamos que aproximadamente un 98 % de las peticiones aleatorias generadas para la operación de búsqueda de la API de YouTube eran inválidas, ya que violaban una o más de las 16 dependencias inter-parámetro presentes en dicha operación [12].

En un trabajo anterior estudiamos cómo predecir automáticamente la validez de las peticiones, desconociendo las dependencias de una API [14]. En este artículo, proponemos una estrategia para la inferencia automática de dichas dependencias, lo que permite poner a prueba una API de manera eficiente (sin consumir más peticiones HTTP de las necesarias) y efectiva (generando peticiones válidas que ejerciten su funcionalidad). En concreto, nuestra estrategia se basa en analizar la especificación OAS de la API y las peticiones y respuestas de la misma para identificar *dependencias potenciales*, validadas posteriormente al realizar nuevas peticiones. Durante todo el proceso, las dependencias se van *refinando*, siendo cada vez más cercanas a las dependencias reales, al mismo tiempo que la API se prueba de forma más exhaustiva progresivamente.

2. Preliminares

En esta sección, introducimos conceptos básicos para contextualizar nuestra propuesta: APIs REST, dependencias inter-parámetro, y pruebas de APIs.

2.1. APIs REST

Las APIs REST [5] proporcionan un mecanismo estándar para manejar recursos por medio de operaciones de lectura, escritura, actualización y eliminación (operaciones *CRUD*). Un recurso es cualquier tipo de información a la que se puede acceder a través de la red, ya sea texto, imágenes o información específica de dominio (p. ej., una canción en la API de Spotify), entre otros. Dichos recursos son accesibles mediante peticiones HTTP (típicamente GET, POST, PUT y DELETE) a rutas determinadas. Así, por ejemplo, para buscar vídeos en la API de YouTube, habría que enviar una petición HTTP GET a la ruta `/youtube/v3/videos`.

[1] `https://swagger.io/docs/specification/describing-parameters/`

Tabla 1: Extracto de la especificación de la API de GitHub.

Parámetro	Tipo	Descripción
visibility	string	Can be one of `all`, `public`, or `private`.
affiliation	string	Comma-separated list of values. Can include: * `owner`: repositories owned by the authenticated user. * `collaborator`: repositories the user has been added to as a collaborator. * `organization_member`: repositories the user has access to through being a member of an organization.
type	string	Can be one of `all`, `owner`, `public`, `private`, `member`. Default: `all`. Will cause a 422 error if used in the same request as `visibility` or `affiliation`.

Las APIs REST pueden describirse con lenguajes formales como OAS, considerado el estándar en la industria. Una especificación OAS describe una API REST en términos de los elementos que la componen: rutas de acceso a los recursos, operaciones y parámetros disponibles, y respuestas, incluyendo códigos de estado y formato del cuerpo (p. ej., un objeto JSON).

2.2. Dependencias inter-parámetro

Una dependencia inter-parámetro es una restricción que limita las posibles combinaciones entre dos o más parámetros de una operación de una API. Dicha restricción deben satisfacerse para generar una petición válida a la API. La Tabla 1 muestra un ejemplo de dos dependencias inter-parámetro en la API de GitHub. Concretamente, el parámetro `type` no puede combinarse ni con `visibility` ni con `affiliation`. En nuestro trabajo previo [11], analizamos la documentación de más de 2.500 operaciones de 40 APIs industriales y concluimos que las dependencias inter-parámetro son extremadamente comunes, estando presentes en 4 de cada 5 APIs. Además, encontramos más de 600 dependencias, las cuales clasificamos en un catálogo de ocho patrones. Debido a la falta de soporte para especificar estas dependencias en lenguajes como OAS, en un trabajo posterior [13], creamos un lenguaje específico de dominio (DSL) para suplir esta carencia, denominado *Inter-parameter Dependency Language* (IDL). IDL permite expresar los ocho patrones de dependencias identificados en nuestro trabajo previo [11]. A continuación, se muestra un listado con la sintaxis de cada tipo de dependencia en IDL.

```
IF videoDefinition THEN type=='video';                          // Requires
Or(query, type);                                                // Or
ZeroOrOne(radius, rankby=='distance');                          // ZeroOrOne
AllOrNone(location, radius);                                    // AllOrNone
OnlyOne(amount_off, percent_off);                               // OnlyOne
publishedAfter >= publishedBefore;                              // Relational
limit + offset <= 1000;                                         // Arithmetic
IF intent=='browse' THEN OnlyOne(ll AND radius, sw AND ne);     // Complex
```

Listing 1: Ejemplos de dependencias IDL extraídos de APIs reales.

2.3. Pruebas de APIs con dependencias inter-parámetro

Probar una API REST implica generar peticiones HTTP y comprobar la validez de las respuestas. Para automatizar este proceso, numerosas técnicas [2,3,4,9,19] derivan casos de prueba a partir de la especificación OAS de la API. Sin embargo, dado que dicha especificación no incluye las dependencias inter-parámetro, es común que las peticiones generadas sean inválidas, al violar alguna de las dependencias. En nuestro trabajo previo [13], creamos IDLReasoner, una herramienta que permite generar llamadas válidas dada una especificación IDL. En este trabajo, vamos un paso más allá y proponemos inferir las dependencias de una API (y generar su especificación IDL) automáticamente.

3. Propuesta

El objetivo de la técnica propuesta es la inferencia de dependencias inter-parámetro simplemente a partir de la especificación de la API y el análisis de sus entradas y salidas. La Figura 1 muestra el diagrama de flujo de la técnica.

En los siguientes apartados, se utilizará a modo de ejemplo las dependencias de GitHub descritas en la Tabla 1. Estas dependencias expresadas en IDL quedan resumidas en `ZeroOrOne(type, visibility)` y `ZeroOrOne(type, affiliation)`, indicando que el parámetro `type` no puede combinarse ni con `visibility` ni con `affiliation`.

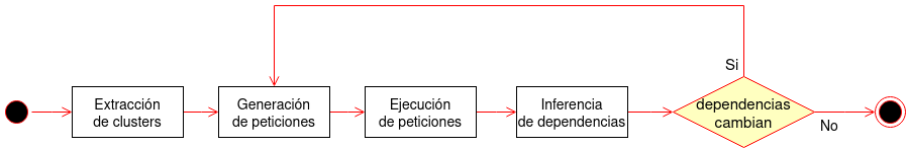

Figura 1: Diagrama de flujo

3.1. Extracción de clusters

El objetivo de la fase inicial es analizar la especificación de la API y determinar los *clusters*. Un cluster es un conjunto de parámetros sobre los que se sospecha que pueda existir una restricción, es decir, candidatos a estar involucrados en alguna dependencia. Esta etapa se compone a su vez de dos pasos fundamentales: el análisis textual y el enriquecimiento.

Análisis textual: este proceso parte de la siguiente suposición: *cuando dos parámetros están relacionados, se espera que en la descripción de al menos uno de los dos se mencione al otro.* De esta forma, a partir de la especificación se construye una matriz de incidencia cuyas filas y columnas son los nombres de los parámetros y el elemento (i, j) es 1 si i y j están relacionados, y 0 si no es así [7]. Por ejemplo, en la descripción del parámetro `type` (Tabla 1) se indica: “Will cause a

Tabla 2: Extracto de la especificación de la API de LanguageTool.

Parámetro	**Tipo**	**Descripción**
enabledRules	string	IDs of rules to be enabled, comma-separated.
disabledRules	string	IDs of rules to be enabled, comma-separated.
enabledCategories	string	IDs of categories to be enabled, comma-separated.
disabledCategories	string	IDs of categories to be enabled, comma-separated.
enabledOnly	string	If `true`, only the rules and categories whose IDs are specified with `enabledRules` or `enabledCategories` are enabled.

422 error if used in the same request as `visibility` or `affiliation`"; en este paso, se detecta que "visibility" y "affiliation" aparecen en la descripción de `type`, construyendo así los clusters {`type, visibility`} y {`type, affiliation`}.

Enriquecimiento: en ocasiones, la premisa de la etapa anterior no se cumple, y parámetros relacionados entre sí no aparecen mutuamente mencionados en sus descripciones. La fase de enriquecimiento ayuda a subsanar esto, añadiendo al cluster nuevos parámetros a fin de completarlo. Para encontrar nuevos candidatos, este paso atiende a otra suposición: *cuando los nombres de dos parámetros son similares, suponemos que están relacionados.* A modo de ejemplo, la Tabla 2 muestra un extracto de la especificación de la API LanguageTool [1] para revisión ortográfica de textos. La operación incluye la dependencia `IF enabledOnly==true THEN NOT (disabledRules OR disabledCategories)`. En las descripciones de `enabledOnly`, `disabledRules` y `disabledCategories` no se mencionan los otros parámetros, pero en la descripción de `enabledOnly` sí se mencionan `enabledRules` y `enabledCategories`. En la fase previa (análisis textual), se extrae el cluster {`enabledOnly`, `enabledRules`, `enabledCategories`}. En esta fase de enriquecimiento, se extrae un cluster adicional: {`enabledOnly`, `enabledRules`, `enabledCategories`, `disabledRules`, `disabledCategories`}. Dos nombres de parámetros se consideran "similares" si su distancia de Levenshtein [10] es menor a cierto umbral configurable.

3.2. Generación de peticiones

Tras la extracción de clusters comienza un proceso iterativo para inferir las dependencias. Al generar las peticiones (casos de prueba) en una iteración determinada, se utilizan las dependencias potenciales conocidas por el sistema hasta ese momento, y dichas peticiones se etiquetan con la *validez esperada.* Una petición es válida si devuelve un código de "éxito" (2XX) e inválida si devuelve un código de "error de cliente" (4XX) [8]. Gracias a IDLReasoner (nuestra herramienta de análisis de especificaciones IDL) [13], es posible generar peticiones válidas e inválidas de acuerdo a unas dependencias determinadas. Por ejemplo, supongamos que en un determinado momento el sistema ha inferido que `IF type=='private' THEN NOT visibility=='private'`. Entonces, se podrá generar una petición que se espera válida con la combinación

`<type=‘private’, visibility=‘all’>` y una inválida con `<type=‘private’, visibility=‘private’>`.

3.3. Ejecución de peticiones

En este paso se invoca a la API con las llamadas generadas anteriormente, y se etiqueta cada una de ellas con su *validez real*, que puede coincidir o no con la validez esperada. La validez se determina en función del código de estado de la respuesta, tal como se ha explicado en el apartado anterior. Por ejemplo, las dos peticiones generadas anteriormente obtendrían códigos de estado 4XX y serían clasificadas como inválidas. Esto se debe a que ambas incumplen la dependencia `ZeroOrOne(type, visibility)`. Nótese que únicamente se esperaba que fuese inválida la segunda petición, sin embargo la primera también lo es.

3.4. Inferencia de dependencias

Este paso toma como entrada el conjunto de peticiones etiquetadas con su validez real y produce como salida un conjunto de dependencias que se ajuste a dicho dataset correctamente etiquetado. Aquí se concentra el mayor aporte lógico de la solución. Consiste en tres pasos que se ejecutan *sobre cada cluster*: *agrupación*, *clasificación* y *traducción*.

Agrupación: en primer lugar, se agrupan las peticiones según las diferentes combinaciones de pares clave-valor de los parámetros del cluster bajo estudio. Siguiendo el ejemplo, se agrupan todas aquellas peticiones que contengan combinaciones tales como `<type=‘private’, visibility=‘all’>`, `<type=‘private’, visibility=‘private’>`, `<type=‘private’>`, etc.

Clasificación: para cada combinación $\mathcal{C}$ resultante del paso anterior, se comprueba si existe al menos una petición válida que presente $\mathcal{C}$:

- Si existe: entonces se sabe que $\mathcal{C}$ es válida, ya que, de no ser así, no podría haberse dado ninguna petición válida con $\mathcal{C}$.
- Si no existe: entonces suponemos que $\mathcal{C}$ es inválida. A diferencia del caso contrario, no sabemos con seguridad que $\mathcal{C}$ sea inválida, ya que es posible que las dependencias que se incumplen en todas las peticiones que presentan $\mathcal{C}$ sean independientes de los parámetros incluidos en $\mathcal{C}$.

Por ejemplo, para `<type=‘private’>` se espera que exista al menos una llamada válida. Por otro lado, para `<type=‘private’, visibility=‘all’>` y `<type=‘private’, visibility=‘private’>` no se encontrará ninguna, por lo que se asume por el momento que esta combinación es inválida.

Traducción: una vez se han identificado todas las combinaciones que se suponen inválidas, se traduce cada combinación a dependencias en formato IDL. Para ello, se elaboran dependencias del tipo `IF a THEN NOT b`, siendo `a` todos los pares clave-valor de la combinación inválida menos uno, que se situará en `b`. De esta forma, se considera que una petición con dicha combinación es inválida.

Tabla 3: Peticiones del ejemplo en dos iteraciones.

ID	type	visibility	affiliation	sort	direction	válida
			Primera iteración:			
0	'private'	'all'	'collaborator'	-	'asc'	No
1	-	'private'	'collaborator'	'created'	-	Sí
2	'public'	-	'owner'	-	-	No
			Segunda iteración:			
3	'owner'	-	-	'pushed'	'desc'	Sí
4	'public'	-	-	'created'	-	Sí
5	'public'	'private'	'collaborator'	-	'asc'	No

Por ejemplo, a partir de las combinaciones inválidas `<type='private', visibility='all'>` y `<type='private', visibility='private'>` se inferirían las dependencias `IF type=='private' THEN NOT visibility=='all'` y `IF type =='private' THEN NOT visibility=='private'`.

A partir de las peticiones etiquetadas con su validez esperada y validez real, se calcula la precisión obtenida en cada iteración. Esto permite medir la bondad de las dependencias inferidas, que irá mejorando en cada iteración mediante un proceso evolutivo. Por ejemplo, la llamada `<type='private', visibility='all'>` se esperaba válida y ha sido inválida, mientras que la llamada `<type='private', visibility='private'>` ha sido inválida, como se predijo. Para esas dos peticiones, la precisión sería pues de un 50 %.

3.5. Ejemplo en dos iteraciones

A continuación, mostramos un ejemplo de funcionamiento de la técnica en dos iteraciones. El ejemplo se refiere a la operación de GitHub mostrada en la Tabla 1.

Extracción de clusters: En la descripción del parámetro `type` aparecen las palabras "visibility" y "affiliation", por lo que se construyen los clusters {`type, visibility`} y {`type, affiliation`}.

Primera iteración: En principio, el sistema no conoce ninguna dependencia, por lo que asume que cualquier llamada es válida. Supongamos que las peticiones generadas en la primera y segunda iteración son las que se muestran en la Tabla 3.

Para el cluster {`type`, `visibility`}, las combinaciones que aparecen son: 0) `<type='private', visibility='all'>`, 1) `<visibility='private'>`, y 2) `<type= 'public'>`, etiquetadas como inválida, válida e inválida, respectivamente. A la vista de estas peticiones, *lo único que el sistema puede aprender* es que la combinación `<visibility='private'>` es válida, ya que la dependencia `ZeroOrOne(type, visibility)` permite que se utilice uno de los parámetros, `type` o `visibility`. En cambio, el sistema *solo puede suponer* que las combinaciones encontradas en las peticiones inválidas, `<type='private', visibility='all'>` y `<type='public'>`, están prohibidas. En el primer caso es cierto, mientras que

en el segundo no, la dependencia que se está incumpliendo es `ZeroOrOne(type, affiliation)`, ajena al cluster {`type`, `visibility`}.

Como resultado, el sistema aprende las dependencias: 1) `IF type==‘private’ THEN NOT visibility==‘all’` y 2) `IF type==‘public’ THEN visibility`; la primera es un verdadero positivo, mientras que la segunda es un falso positivo.

Segunda iteración: Ahora el sistema *aprende* que `<type=‘owner’>` y `<type=‘public’>` son válidas, mientras que *supone* que `<type=‘public’, visibility=‘private’>` es inválida. La combinación `<type=‘public’>` se supuso como inválida en la iteración anterior; sin embargo, en esta iteración se ha demostrado lo contrario (hay una petición válida que la contiene), por lo que la dependencia inferida anteriormente se elimina. Tras la segunda iteración, el sistema ha inferido las dependencias: 1) `IF type==‘private’ THEN NOT visibility==‘all’` y 2) `IF type==‘public’ THEN NOT visibility==‘private’`, ambas verdaderos positivos.

4. Evaluación

Para la evaluación de la solución que se propone, se dará respuesta a las siguientes preguntas de investigación (PIs):

- **PI1:** *¿Qué porcentaje de peticiones correctas pueden generarse a) ignorando las dependencias, y b) mediante la solución propuesta?* Se entiende que una petición es correcta cuando la validez obtenida al invocar la API es la misma que la esperada atendiendo a las dependencias conocidas.
- **PI2:** *¿Con cuántas llamadas se estabiliza el conjunto de dependencias inferidas?*

4.1. Implementación

La parte lógica de evolución e inferencia de dependencias ha sido implementada en `Python`, mientras que la generación e invocación de peticiones son llevadas a cabo por las herramientas RESTest e IDLReasoner. RESTest [12] es un framework de pruebas de caja negra para APIs REST. IDLReasoner [13] está integrado en RESTest y permite la generación automática de peticiones válidas e inválidas de acuerdo a una especificación IDL determinada.

4.2. Resultados experimentales

Hemos evaluado la propuesta con 4 servicios RESTful de APIs industriales: obtención de los repositorios de un usuario en GitHub [6], comprobación ortográfica de texto en LanguageTool [1], creación de un nuevo cupón en Stripe [18] y lectura de comentarios en YouTube [20]. La documentación de la API de LanguageTool define el parámetro `language` como una cadena de texto libre, aunque en realidad se trata de un enumerado (solo puede tomar un conjunto finito de valores). Además, hay una dependencia que restringe un valor en

Tabla 4: Resultados experimentales.

API	Precisión usando llamadas aleatorias (%)	Precisión con las dependencias inferidas (%)	Nº de peticiones
GitHub	60.0	100	300
Stripe	41.8	100	50
YouTube	38.8	95.6	250
LanguageTool	35.6	66.8	800
LanguageTool*	35.6	98.2	1750
Media	**44.1**	**90.6**	**350**
Media[1]	**42.6**	**92.1**	**630**

[1] media con LanguageTool*

concreto que dicho parámetro puede tomar: `IF preferredVariants THEN language=='auto'`. Ante esta ambigüedad, evaluamos el servicio de LanguageTool tanto con la documentación original como con la documentación manualmente modificada, considerando `language` un parámetro enumerado; este último caso se recoge bajo el nombre de LanguageTool*.

Para cada servicio evaluado, generamos peticiones usando dos estrategias: a) aleatoriamente (sin tener en cuenta las dependencias); y b) utilizando las dependencias inferidas automáticamente. En cada iteración del aprendizaje, se generan 50 peticiones, un 70 % válidas y un 30 % inválidas. La Tabla 4 muestra los resultados obtenidos. La última columna denota el número de peticiones necesarias para inferir las dependencias (cuando termina el aprendizaje). Como se puede observar, los resultados son prometedores. En dos servicios (GitHub y Stripe) se consiguen inferir las dependencias exactas (100 % de precisión). En LanguageTool se obtiene una precisión sensiblemente menor, dada la ambigüedad comentada anteriormente. Sin embargo, en LanguageTool* (considerando el parámetro `language` como enumerado) la precisión obtenida se acerca al 100 %. En base a los resultados obtenidos, podemos responder a las PIs como sigue:

PI1: *el porcentaje de peticiones correctas obtenidas utilizando la solución propuesta es de un 90.6 %, más del doble que con llamadas aleatorias (44.1 %).*
PI2: *el número medio de peticiones necesarias para que el aprendizaje finalice es 350 llamadas.*

5. Conclusión y trabajo futuro

En este artículo, hemos propuesto una técnica de inferencia de dependencias inter-parámetro en APIs REST que permite mejorar la generación de casos de prueba en este dominio. Las dependencias se infieren con un enfoque de caja negra, utilizando únicamente la especificación de la API y analizando sus entradas y salidas. Gracias a esta técnica, se puede obtener un porcentaje de llamadas correctas de un 90 % aproximadamente. El trabajo futuro irá dirigido a tres líneas

principales: minimización de las dependencias inferidas, mejora de la eficiencia de la propuesta, y extensión de la evaluación con nuevos servicios web.

Referencias

1. Languagetool documentation, `https://languagetoolplus.com/http-api/#!/default/post_check`, accessed May 2021
2. Arcuri, A.: RESTful API Automated Test Case Generation with EvoMaster. ACM Transactions on Software Engineering and Methodology **28**(1), 1–37 (2019)
3. Atlidakis, V., Godefroid, P., Polishchuk, M.: RESTler: Stateful REST API Fuzzing. In: International Conference on Software Engineering. pp. 748–758 (2019)
4. Ed-douibi, H., Izquierdo, J.L.C., Cabot, J.: Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In: IEEE International Enterprise Distributed Object Computing Conference. pp. 181–190 (2018)
5. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
6. GitHub API, `https://developer.github.com/v3/`, accessed May 2021
7. Grent, H., Akimov, A., Aniche, M.: Automatically identifying parameter constraints in complex web apis: A case study at adyen (2021)
8. Jacobson, D., Brail, G., Woods, D.: APIs: A Strategy Guide. O'Reilly Media, Inc. (2011)
9. Karlsson, S., Causevic, A., Sundmark, D.: QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In: IEEE International Conference on Software Testing, Validation and Verification. pp. 131–141 (2020)
10. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10**, 707 (Feb 1966)
11. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs. In: International Conference on Service-Oriented Computing. pp. 399–414 (2019)
12. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In: International Conference on Service-Oriented Computing. pp. 459–475 (2020)
13. Martin-Lopez, A., Segura, S., Müller, C., Ruiz-Cortés, A.: Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. IEEE Transactions on Services Computing (2020), in press
14. Mirabella, A.G., Martin-Lopez, A., Segura, S., Valencia-Cabrera, L., Ruiz-Cortés, A.: Deep Learning-Based Prediction of Test Input Validity for RESTful APIs. In: International Workshop on Testing for Deep Learning and Deep Learning for Testing (2021)
15. OpenAPI Specification, `https://www.openapis.org`, accessed May 2021
16. ProgrammableWeb API Directory, `http://www.programmableweb.com/`, accessed May 2021
17. Segura, S., Parejo, J.A., Troya, J., Ruiz-Cortés, A.: Metamorphic Testing of RESTful Web APIs. IEEE Transactions on Software Engineering **44**(11), 1083–1099 (2018)
18. Stripe API, `https://stripe.com/docs/api`, accessed May 2021
19. Viglianisi, E., Dallago, M., Ceccato, M.: RestTestGen: Automated Black-Box Testing of RESTful APIs. In: IEEE International Conference on Software Testing, Validation and Verification. pp. 142–152 (2020)
20. YouTube API, `https://developers.google.com/youtube/v3/docs`, accessed May 2021