ICSE: G: Automated Management of Inter-Parameter Dependencies in Web APIs

Alberto Martin-Lopez (Advisors: Sergio Segura, Antonio Ruiz-Cortés)
SCORE Lab, I3US Institute, Universidad de Sevilla
Seville, Spain
alberto.martin@us.es

ABSTRACT

Web services often impose inter-parameter dependencies that restrict the way in which two or more input parameters can be combined to form valid calls to the service. Unfortunately, current specification languages for web services like the OpenAPI Specification (OAS) provide no support for the formal description of such dependencies, which makes it hardly possible to automatically discover and interact with services without human intervention. In this work, we present an approach for the automated management of inter-parameter dependencies in web APIs. Specifically, we present a domain-specific language for the description of these dependencies, a mapping to translate them into a constraint satisfaction problem and a catalog of automated analysis operations. The approach is supported by a comprehensive tool suite. We show the potential of our contributions in the domain of automated testing of web APIs. We uncovered over 4K failures and dozens of bugs in APIs such as YouTube and Yelp. Beyond testing, our approach paves the way for a new generation of specification-driven applications in areas such as API monitoring and code generation.

1 PROBLEM AND MOTIVATION

Web APIs allow applications to talk to each other over the network. The widespread use of web APIs is reflected in the size of popular API repositories such as ProgrammableWeb [8], currently indexing over 24K web APIs. Companies such as Facebook, Twitter, Google or Netflix receive billions of API calls everyday from thousands of different third-party applications and devices, which constitutes more than half of their total traffic [22]. Modern web APIs usually follow the REST architectural style [20], being referred to as RESTful web APIs. RESTful APIs can be described using languages such as the OpenAPI Specification (OAS) [7], which has arguably become the industry standard. An OAS document describes a RESTful API in terms of the elements it consists of, namely paths, operations, resources, request parameters and API responses. It is both human- and machine-readable, making it possible to automatically generate, for instance, documentation, source code, or basic test cases.

Web APIs often present *inter-parameter dependencies*, i.e., constraints that restrict the way in which two or more input parameters can be combined to form valid calls to the service. For instance, in the search operation of the YouTube API [15], when using the parameter `videoDefinition` (e.g., to search videos in high definition) the `type` parameter must be set to `'video'`, otherwise an HTTP 400 status code ("Bad Request" client error) is returned. Current API specification languages such as OAS [7] or RAML [10] provide no support for the description of these dependencies. For example, the OAS documentation states:[1] "*OpenAPI 3.0 does not support parameter dependencies and mutually exclusive parameters. (...) What you can do is document the restrictions in the parameter description and define the logic in the 400 Bad Request response*". This makes it hardly possible to interact with the services without human intervention. For example, it would be extremely difficult, possibly infeasible, to automatically generate test cases for the YouTube API without an explicit and machine-readable definition of its dependencies. Industry has shown great interest in this issue, as reflected in an open feature request in OAS entitled "Support interdependencies between query parameters", created in January 2015. To date, it has received over 370 votes, becoming the most upvoted issue of all times in the repository, and it has received 65 comments from 37 participants [13].

To tackle the problem of handling inter-parameter dependencies in web APIs, we first performed an exhaustive review of over 2.5K operations from 40 industrial web APIs, in order to get a solid grasp of how these dependencies emerge in practice [25]. We found over 600 dependencies, which we classified into a catalog of seven patterns consistently found in real-world APIs. This catalog served as the basis for the design of Inter-parameter Dependency Language (IDL), a domain-specific language (DSL) for the description of inter-parameter dependencies in web APIs. We also devised a constraint programming-aided tool, IDLReasoner, that translates IDL dependencies into a constraint satisfaction problem (CSP) and supports multiple analysis operations, e.g., to automatically generate a valid request (i.e., a request satisfying all dependencies) or to check whether a given API request is valid.

To illustrate one of the many potential applications of our approach, we compared the effectiveness of random testing and IDLReasoner in generating valid requests (i.e., those satisfying all inter-parameter dependencies) and detecting failures in 10 services from 6 commercial APIs. As expected, random testing struggled to generate valid requests, whereas IDLReasoner generated 100% valid requests for all the services under test. More importantly, test cases generated with IDLReasoner revealed 48% more failures (4,332 vs. 2,920) and 18 more unique bugs (32 vs. 14) than random testing. Other promising applications of our work include continuous web API monitoring or automatic code generation of client software development kits (SDKs) and server stubs, among others.

2 BACKGROUND AND RELATED WORK

The management of inter-parameter dependencies in web APIs has predominantly been studied from three different perspectives: the specification of dependencies, their analysis, and their inference.

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

2.1 Specification

Oostvogels et al. [29] proposed OAS-IP, a DSL for the description of inter-parameter constraints in OAS. They studied six commercial APIs and designed a DSL supporting all the types of constraints they found, namely three. RAML [10] supports the specification of mutually exclusive parameters (i.e., only one out of several parameters must be used), but it lacks support for other types of dependencies such as conditionally required parameters or arithmetic dependencies [25]. Previous authors proposed using XML-based languages such as CLiX [1, 18] or CxWSDL [30] to specify constraints in other types of web services such as WSDL [14] or OWL-S [11], technologies increasingly in disuse nowadays. Our research in the specification of dependencies stands out mainly in two ways. First, IDL (the DSL we propose) is specification-independent, therefore it can be integrated into any web API description language. Second, IDL is based on a thorough analysis of industrial APIs, and it supports seven dependency patterns consistently found in practice, most of which are not supported by previous proposals.

2.2 Analysis

The analysis of inter-parameter dependencies in web APIs has somewhat been overlooked in the literature, essentially because current web API specification languages [7, 10] do not support them. For instance, current automated testing approaches for web APIs [16, 17, 19, 23, 31] do not handle inter-parameter dependencies and therefore show limited applicability in practice. Previous languages such as CLiX [18] and CxWSDL [30] could be used to generate requests satisfying and violating the constraints specified with them, but they do not support most of the dependencies from our catalog [25], and they are meant for legacy technologies [11, 14]. Our approach for analyzing inter-parameter dependencies automatically goes beyond the testing domain, as we propose a catalog of analysis operations that prove useful for multiple applications such as web API development and monitoring.

2.3 Inference

Wu et al. [32] presented an approach for the automated inference of dependency constraints among input parameters in web services. They first established six dependency patterns, and then leveraged the service documentation, the SDK and the service itself to extract and validate candidates of dependencies. Grent et al. [21] proposed a similar approach, leveraging the service implementation instead of the SDK. These techniques assume access to the source code, either the system or the SDK, resources that are not always available for web APIs, especially the source code of the system. We are currently working on the automated inference of inter-parameter dependencies using a purely black-box approach, with the application of artificial intelligence (AI) techniques [28].

3 APPROACH AND UNIQUENESS

In this section, we explain our proposed approach for the automated management of inter-parameter dependencies in web APIs.

3.1 Catalog of Dependencies

Addressing the problem of modeling inter-parameter dependencies in web APIs should necessarily start by understanding how these dependencies emerge in practice. Inspired by this idea, in previous work [25], we performed a thorough study on the presence of inter-parameter dependencies in web APIs, involving 2,557 operations from 40 industrial APIs. We found that dependencies are extremely common and pervasive, being present in about 4 out of every 5 APIs, across all application domains and types of operations. As the main outcome of our study, we classified all the dependencies found (633) into a catalog of seven different patterns, described below. For the sake of simplicity, dependencies are described using single parameters, however, all dependencies can be generalized to consider groups of parameters using conjunctive and disjunctive connectors. Moreover, dependencies can affect not only the presence or absence of parameters, but also the values that they can take. We identified the following patterns:

Requires. The presence of a parameter p_1 in an API call requires the presence of another parameter p_2. As previously mentioned, p_1 and p_2 can be generalized to groups of parameters and parameter-value relations, e.g., if parameter p_1 is present in the API call, then parameter p_2 must be present too, and parameter p_3 must be equal to value v. As an example, in the GitHub API, when creating a card in a project, if the parameter `content_id` is present, then `content_type` becomes required.

Or. Given a set of parameters $p_1, p_2, \ldots, p_n$, one or more of them must be included in the API call. For instance, when setting the information of a photo in the Flickr API, at least one of the parameters `title` or `description` must be provided.

OnlyOne. Given a set of parameters $p_1, p_2, \ldots, p_n$, one, and only one of them must be included in the API call. For example, in the Last.fm API, when getting the information about an artist, this can be identified with two possible parameters, `artist` or `mbid`, and only one must be used.

AllOrNone. Given a set of parameters $p_1, p_2, \ldots, p_n$, either all of them are provided or none of them. In the Yelp API, for example, when searching for businesses, the location can optionally be specified with two parameters, `latitude` and `longitude`, which must be used together.

ZeroOrOne. Given a set of parameters $p_1, p_2, \ldots, p_n$, zero or one can be present in the API call. As an example, in the YouTube API, when searching for videos, either the `id` or the `maxResults` parameters may be specified to filter results, but not both at the same time.

Arithmetic/Relational. Given a set of parameters $p_1, p_2, \ldots, p_n$, they are related by means of arithmetic and/or relational constraints, e.g., $p_1 + p_2 < 100$. For instance, in the Twitter API, when searching for tweets, the `since_id` parameter must be lower than the `max_id` parameter.

Complex. These dependencies involve two or more of the types of constraints previously presented. For example, in the Tumblr API, when creating a new post, if the `type` parameter is set to `'video'`, then either `embed` or `data` must be specified, but not both (this dependency combines the *Requires* and the *OnlyOne* patterns).

3.2 Inter-Parameter Dependency Language

We have proposed Inter-parameter Dependency Language (IDL) [24], a DSL specifically tailored to express the seven types of inter-parameter dependencies identified in our study on real-world APIs [25]

```
Model:
    Dependency*;
Dependency:
    RelationalDependency | ArithmeticDependency |
    ConditionalDependency | PredefinedDependency;
RelationalDependency:
    Param RelationalOperator Param;
ArithmeticDependency:
    Operation RelationalOperator DOUBLE;
Operation:
    Param OperationContinuation |
    '(' Operation ')' OperationContinuation?;
OperationContinuation:
    ArithmeticOperator (Param | Operation);
ConditionalDependency:
    'IF' Predicate 'THEN' Predicate;
Predicate:
    Clause ClauseContinuation?;
Clause:
    (Term | RelationalDependency | ArithmeticDependency
    | PredefinedDependency) | 'NOT'? '(' Predicate ')';
Term:
    'NOT'? (Param | ParamValueRelation);
Param:
    ID | '[' ID ']';
ParamValueRelation:
    Param '==' STRING('|'STRING)* |
    Param 'LIKE' PATTERN_STRING | Param '==' BOOLEAN |
    Param RelationalOperator DOUBLE;
ClauseContinuation:
    ('AND' | 'OR') Predicate;
PredefinedDependency:
    'NOT'? ('Or' | 'OnlyOne' | 'AllOrNone' | 'ZeroOrOne')
    '(' Clause (',' Clause)+ ')';
RelationalOperator:
    '<' | '>' | '<=' | '>=' | '==' | '!=';
ArithmeticOperator:
    '+' | '-' | '*' | '/';
```

Listing 1: Simplified grammar of IDL.

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

Listing 2: IDL dependencies examples from real-world APIs.

and described in the previous section. A simplified version of the grammar of the language is provided in Listing 1—the full version is available as a part of the implementation of IDL [4]. Listing 2 shows examples of each dependency type expressed in IDL, extracted from APIs such as YouTube, Google Maps and Tumblr, among others.

The key elements of the language are terms and predicates. Both of them can evaluate to true or false. A *term* is an atomic element of the language and can be represented by: (1) a parameter name (e.g., `p1`) being evaluated as true if the parameter is set (regardless of the value), or false otherwise; or (2) a parameter-value relation, evaluated as true if the parameter is selected and satisfies the relation (e.g., `p1>=100`). A *predicate* is a combination of one or more terms and dependencies joined by the logical operators NOT, AND, and OR. Parentheses are allowed in order to specify the operator precedence.

3.3 Automated Analysis

We propose translating IDL specifications into constraint satisfaction problems (CSPs) that can be then analysed using state-of-the-art constraint programming tools. A CSP is defined as a 3-tuple (V, D, C) composed of a set of variables V, their domains D and a number of constraints C. A solution for a CSP is an assignment of values to the variables in V from their domains in D so that all the constraints in C are satisfied. Parameter names (V) and domains (D) are extracted from the API specification (e.g., an OAS document), constraints (C) are obtained from the IDL dependencies. Table 1 describes the mapping from IDL to CSP. For each parameter, two CSP variables are created: (1) one representing the parameter itself (p_i), and (2) a Boolean variable to express whether the parameter is set or not ($p_i Set$). If a parameter p_i is required (i.e., it must be present in all API calls), the constraint $p_i Set == true$ is added to the set of constraints C. Rows 2 and 3 of the table express how IDL *terms* are mapped to a CSP. Lastly, IDL *predicates* and *dependencies* are defined recursively using the function $map(E)$, where E is either a term, a predicate or a dependency. This is because complex dependencies can contain nested dependencies and predicates.

Once that inter-parameter dependencies are expressed as a CSP, several analysis operations can be invoked. We have defined ten operations [24], the following are some examples:

- **Valid specification**. This operation takes as input an IDL specification and returns a Boolean indicating whether it is valid or not. An IDL specification is valid if it does not contain inconsistencies (e.g., contradictory dependencies and dead parameters) and it has at least one solution (i.e., a valid request can be generated).
- **Valid request**. This operation takes as input an IDL specification and an API request and returns a Boolean indicating whether the request is valid or not, i.e., whether it satisfies all the inter-parameter dependencies.
- **Dead parameter**. This operation takes as input an IDL specification and a parameter name and returns a Boolean indicating whether the parameter is dead, i.e., it cannot be included in any valid call to the service. Dead parameters are caused by inconsistencies in the service specification.
- **Random request**. This operation takes as input an IDL specification and returns a random valid request.

3.4 Tooling Support

As a part of our contribution, we provide a set of open-source tools supporting the specification and analysis of inter-parameter dependencies in web APIs, namely: (1) a parser for IDL specifications and an (Eclipse-based) editor implementing it, supporting features such as code completion, syntax coloring and error checking [4]; (2) IDL4OAS, an OAS extension supporting the specification of IDL dependencies in OAS, one of the most requested features by the community [13]; and (3) IDLReasoner [3], a CSP-based Java library implementing the ten analysis operations proposed [24], which eases the integration of our approach into any external project.

4 RESULTS AND CONTRIBUTIONS

Our approach enables promising specification-driven applications in areas like web API development, monitoring and testing. Next, we detail our contributions to testing, our main application domain, and we outline other applications as a part of our ongoing work.

4.1 Automated Testing of RESTful APIs

Testing RESTful web APIs involves generating HTTP requests and asserting HTTP responses. Current approaches for automated testing of RESTful web APIs mostly rely on black-box fuzzing: testing

Table 1: IDL to CSP mapping.

API Parameters		CSP Mapping
	[Parameters] P	$\forall p_i \in P, \begin{cases} V \leftarrow V \cup p_i \cup p_iSet \\ D \leftarrow D \cup domain(p_i) \cup Boolean \\ C \leftarrow C \cup p_iSet == true \text{ (if } p_i \text{ is required)} \end{cases}$
IDL Element		**CSP Mapping**
Terms: map(T)	[Parameter] p_i	$C \leftarrow C \cup \{p_iSet == true\}$
	[Parameter-Value Relation] $p_i\ relOp^{\star}\ v$	$C \leftarrow C \cup \{p_i\ relOp\ v \wedge\ p_iSet == true\}$
Predicates: map(P)	[Term] T	$map(T)$
	[Dependency] D	$map(D)$
	[Term AND Predicate] T AND P	$C \leftarrow C \cup map(T) \wedge map(P)$
	[Term OR Predicate] T OR P	$C \leftarrow C \cup map(T) \vee map(P)$
	[NOT Predicate] NOT P	$C \leftarrow C \cup \neg map(P)$
Dependencies: map(D)	[Requires] IF P_i THEN P_j	$C \leftarrow C \cup map(P_i) \implies map(P_j)$
	[Or] $Or(P_1, ..., P_n)$	$C \leftarrow C \cup \bigvee_{i=1}^{n} map(P_i)$
	[OnlyOne] $OnlyOne(P_1, ..., P_n)$	$C \leftarrow C \cup \{\forall_{i=1}^{n}, \forall_{j=1}^{n} \mid i \neq j, map(P_i) \implies \neg map(P_j)\}$
	[AllOrNone] $AllOrNone(P_1, ..., P_n)$	$C \leftarrow C \cup \forall_{i=1}^{n}, \forall_{j=1}^{n} \mid i \neq j, \{map(P_i) \implies map(P_j)\} \wedge \{\neg map(P_i) \implies \neg map(P_j)\}$
	[ZeroOrOne] $ZeroOrOne(P_1, ..., P_n)$	$C \leftarrow C \cup \{map(OnlyOne(P_1, ..., P_n))\} \vee \{\bigwedge_{i=1}^{n} \neg map(P_i)\}$
	[Relational Dependency] $p_i\ relOp^{\star}\ p_j$	$C \leftarrow C \cup \{(p_iSet == true \wedge p_jSet == true) \implies p_i\ relOp\ p_j\}$
	[Arithmetic Dependency] $p_i\ arOp^{\diamond}\ p_j\ arOp ... p_n\ relOp^{\star}\ v$	$C \leftarrow C \cup \{(p_iSet == true \wedge p_jSet == true \wedge\ ... p_nSet == true) \implies (p_i\ arOp\ p_j\ arOp\ ...\ p_n\ relOp\ v)\}$

$\star$ $relOp = \{< \mid == \mid \neq \mid \geq \mid \leq \mid >\}$
$\diamond$ $arOp = \{+ \mid - \mid * \mid \div\}$

Table 2: Subject APIs and evaluation results.

API	Operation	Parameters	Dependencies	Parameters in dependencies	Valid requests		Failures	
					Random	IDL	Random	IDL
Foursquare	Search venues	17	8	10 (59%)	89.0%	100%	1,169	1,464
GitHub	Get user repositories	5	2	3 (60%)	62.1%	100%	487	236
Stripe	Create coupon	9	3	5 (56%)	17.1%	100%	0	180
Stripe	Create product	18	6	11 (61%)	1.3%	100%	0	535
Tumblr	Get blog likes	5	1	3 (60%)	65.5%	100%	1,195	1,063
Yelp	Search businesses	14	3	7 (50%)	54.6%	97.1%	67	161
Yelp	Search transactions	4	1	3 (75%)	60.1%	94.9%	0	54
YouTube	Get comment threads	11	5	8 (73%)	20.5%	99.9%	0	10
YouTube	Get videos	12	5	7 (58%)	49.2%	100%	2	116
YouTube	Search	31	16	25 (81%)	1.6%	100%	0	513
Total					42.1%	99.2%	2,920	4,332

the services with random requests conforming to the OpenAPI specification of the API [17, 19, 23, 31]. However, these approaches do not support inter-parameter dependencies since, as previously mentioned, these are not formally described in the API specification used as input. As a result, existing approaches simply ignore dependencies and resort to brute force to generate valid requests, i.e., those satisfying all input constraints. This is not only extremely inefficient, but it is also unlikely to work for most real-world services, where inter-parameter dependencies are complex and pervasive. The IDL tool suite can nicely complement existing test data generators for RESTful web APIs, enabling the automated generation of API requests satisfying all the inter-parameter dependencies.

We report the results of two experiments on 10 RESTful services comparing random test case generation, where inter-parameter dependencies are ignored, and IDLResoner, supporting the automated management of inter-parameter dependencies described in IDL. For the implementation of both strategies, we used RESTest [26], an open-source testing framework for RESTful web APIs primarily developed by the author and available on GitHub [9]. In the first experiment, we generated 1,000 valid requests with both test generation strategies and compared the ratio of valid requests actually generated (i.e., those returning a success HTTP status code). In the second experiment, we generated 2,000 requests, both valid and invalid (e.g., using invalid parameter values or violating dependencies), and compared the total number of failures observed. Failures can occur for multiple reasons, namely: 5XX status codes (i.e., server errors), mismatches with the API specification (e.g., a required property is not present in the response body), or unexpected status codes (e.g., obtaining a 400 "Bad Request" status code after a valid request). Note that this last type of failure cannot be detected when ignoring inter-parameter dependencies, since this is required to know whether a request is valid or not.

Table 2 depicts the RESTful services under test and the evaluation results. For each service, the table shows the API name,

operation tested, number of input parameters, number of IDL dependencies, number (and percentage) of different parameters involved in at least one dependency, and the results obtained. As illustrated, the random strategy generated 42.1% valid requests on average. There were extreme cases like the APIs of YouTube and Stripe, where barely 1% of the requests generated were valid, due to the high number and complexity of the dependencies. In contrast, with IDLReasoner, all requests generated were valid. Some requests did not obtain successful responses (e.g., in Yelp), but this was due to actual faults in the services (e.g., when returning a 500 "Internal Server Error" status code). In terms of failures, thanks to IDLReasoner we uncovered 48% more failures than the random strategy (4,332 vs. 2,920). Furthermore, while the random technique detected only failures related to 5XX status codes and mismatches with the API specification, IDLReasoner unveiled more complex failures such as dependencies not described in the documentation of the services and unexpected error messages. We analyzed all the failures obtained and triaged them into 32 unique bugs. More details are available in the supplementary material of this paper [12] and in our previous work [26].

4.2 Other Applications

The potential applications of our research spread beyond the testing domain. Next we describe some of the additional lines of research that we are currently working on.

Automatic code generation. Given an API specification enriched with IDL dependencies (e.g., using our IDL4OAS extension), our IDL parser [4] could drive the generation of built-in assertions in the automatically generated code of client SDKs and server stubs. We are in the process of integrating this feature into the popular code generation framework OpenAPI Generator [6].

Dependency-aware API gateways. We are currently implementing our approach in Kong [5], a popular API gateway. By leveraging the *Valid request* operation from IDLReasoner, the gateway will be able to automatically reject invalid requests violating any dependencies, without even redirecting the call to the corresponding service, saving time and user quota.

Automated inference of dependencies. We are applying deep learning techniques for automatically inferring inter-parameter dependencies in web APIs with a purely black-box approach [28].

4.3 Research Impact

My research on inter-parameter dependencies in web APIs has led to **four first-author publications** and one co-authored publication, including: a paper in *IEEE TSC* [24], two papers at *ICSOC* [25, 26], a book chapter [27], and a workshop paper [28]. Our approach has been integrated into mature testing tools for RESTful web APIs such as RESTest [9] and EvoMaster [2], the latter during a research stay with Professor Andrea Arcuri. We have found bugs in a completely automated fashion in commercial APIs with millions of users worldwide such as Foursquare, Yelp and YouTube [26]. We have been approached by the OAS steering committee [7] and other side project developers [6] to integrate our approach into the standard. Our proposal has the most positive votes (over 80) in the OAS GitHub repository [13]. The results obtained so far have allowed me to obtain a **Fulbright grant**[2] (only 13 awarded in Spain, 2 for the engineering field) to conduct research for six months at the University of California, Berkeley (June - December, 2021), under the supervision of Professor Armando Fox, where we aim to exploit our results in world-leading software companies.

REFERENCES

[1] [n.d.]. *CLiX - A Validation Rule Language for XML.* https://www.w3.org/2004/12/rules-ws/paper/24/ accessed April 2021.
[2] [n.d.]. *EvoMaster: A Tool for Automatically Generating System-Level Test Cases.* https://github.com/EMResearch/EvoMaster accessed April 2021.
[3] [n.d.]. *IDLReasoner.* https://github.com/isa-group/IDLReasoner accessed April 2021.
[4] [n.d.]. *Inter-parameter Dependency Language (IDL).* https://github.com/isa-group/IDL accessed April 2021.
[5] [n.d.]. *Kong API Gateway.* https://konghq.com/kong/ accessed April 2021.
[6] [n.d.]. *OpenAPI Generator.* https://github.com/OpenAPITools/openapi-generator accessed April 2021.
[7] [n.d.]. *OpenAPI Specification.* https://www.openapis.org accessed April 2021.
[8] [n.d.]. *ProgrammableWeb.* https://www.programmableweb.com accessed April 2021.
[9] [n.d.]. *RESTest: Automated Black-Box Testing of RESTful Web APIs.* https://github.com/isa-group/RESTest accessed April 2021.
[10] [n.d.]. *RESTful API Modeling Language (RAML).* https://raml.org accessed April 2021.
[11] [n.d.]. *Semantic Markup for Web Services (OWL-S).* https://www.w3.org/Submission/OWL-S/ accessed April 2021.
[12] [n.d.]. *Supplementary material of the paper.* https://github.com/isa-group/acm-src-2021-supplementary-material accessed April 2021.
[13] [n.d.]. *Support interdependencies between query parameters · Issue #256 · OAI/OpenAPI-Specification.* https://github.com/OAI/OpenAPI-Specification/issues/256 accessed April 2021.
[14] [n.d.]. *Web Services Description Language (WSDL) Version 2.0.* https://www.w3.org/TR/wsdl20/ accessed April 2021.
[15] [n.d.]. *YouTube API.* https://developers.google.com/youtube/v3/docs accessed April 2021.
[16] A. Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM TOSEM* 28, 1 (2019), 1–37.
[17] V. Atlidakis, P. Godefroid, and M. Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *ICSE*. 748–758.
[18] D. Cacciagrano, F. Corradini, R. Culmone, and L. Vito. 2006. Dynamic Constraint-based Invocation of Web Services. In *WS-FM*. 138–147.
[19] H. Ed-douibi, J.L.C. Izquierdo, and J. Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *EDOC*. 181–190.
[20] R. T. Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation.
[21] H. Grent, A. Akimov, and M. Aniche. 2021. Automatically Identifying Parameter Constraints in Complex Web APIs: A Case Study at Adyen. In *ICSE SEIP*.
[22] D. Jacobson, G. Brail, and D. Woods. 2011. *APIs: A Strategy Guide.* O'Reilly Media, Inc.
[23] S. Karlsson, A. Causevic, and D. Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In *ICST*.
[24] A. Martin-Lopez, S. Segura, C. Müller, and A. Ruiz-Cortés. 2021. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE TSC* (2021).
[25] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. 2019. A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs. In *ICSOC*. 399–414.
[26] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. 2020. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *ICSOC*. 459–475.
[27] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. 2021. *Inter-Parameter Dependencies in Real-World Web APIs: The IDEA Dataset.* CRC Press, Taylor & Francis.
[28] A.G. Mirabella, A. Martin-Lopez, S. Segura, L. Valencia-Cabrera, and A. Ruiz-Cortés. 2021. Deep Learning-Based Prediction of Test Input Validity for RESTful APIs. In *DeepTest*.
[29] Oostvogels, N., De Koster, J., De Meuter, W. 2017. Inter-parameter Constraints in Contemporary Web APIs. In *ICWE*. 323–335.
[30] C.a. Sun, M. Li, J. Jia, and J. Han. 2018. Constraint-Based Model-Driven Testing of Web Services for Behavior Conformance. In *ICSOC*. 543–559.
[31] E. Viglianisi, M. Dallago, and M. Ceccato. 2020. RestTestGen: Automated Black-Box Testing of RESTful APIs. In *ICST*.
[32] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. 2013. Inferring Dependency Constraints on Parameters for Web Services. In *WWW*. 1421–1432.

[2] https://eca.state.gov/fulbright