Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs

Alberto Martin-Lopez, Sergio Segura, Carlos Müller, and Antonio Ruiz-Cortés

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 article, we present an approach for the specification and automated analysis of inter-parameter dependencies in web APIs. We first present a domain-specific language, called *Inter-parameter Dependency Language* (IDL), for the specification of dependencies among input parameters in web services. Then, we propose a mapping to translate an IDL document into a constraint satisfaction problem (CSP), enabling the automated analysis of IDL specifications using standard CSP-based reasoning operations. Specifically, we present a catalogue of seven analysis operations on IDL documents allowing to compute, for example, whether a given request satisfies all the dependencies of the service. Finally, we present a tool suite including an editor, a parser, an OAS extension, a constraint programming-aided library, and a test suite supporting IDL specifications and their analyses. Together, these contributions pave the way for a new range of specification-driven applications in areas such as code generation and testing.

Index Terms—Web API, REST, inter-parameter dependency, DSL, automated analysis.

1 Introduction

Web Application Programming Interfaces (APIs) allow systems to interact with each other over the network, typically using web services [21], [41]. Web APIs are rapidly proliferating as the cornerstone for software integration enabling new consumption models such as mobile, social, Internet of Things (IoT), or cloud applications. Many companies are also exposing their existing assets as private APIs, fostering reusability, integration, and innovation within the boundaries of their own companies [21], [22]. Popular API directories such as ProgrammableWeb [36] and RapidAPI [39] currently index over 23K and 10K web APIs, respectively, from multiple domains such as shopping, finances, social networks, or telephony.

Modern web APIs typically adhere to the REpresentational State Transfer (REST) architectural style, being referred to as RESTful web APIs [10]. *RESTful web APIs* are decomposed into multiple web services, where each service implements one or more create, read, update, or delete (CRUD) operations over a resource (e.g., an invoice in the PayPal API), typically through HTTP interactions. RESTful APIs are commonly described using languages such as the OpenAPI Specification (OAS) [33], originally created as a part of the Swagger tool suite [44], or the RESTful API Modeling Language (RAML) [38]. These languages are designed to provide a structured description of a RESTful web API that allows both humans and computers to discover and understand the capabilities of a service without requiring access to the source code or additional documentation. Once an API is described in an OAS document, for example, the specification can be used to generate documentation, code (clients and servers), or even basic automated test cases [44]. In this article, we focus on RESTful web APIs and OAS as the arguable standards for web integration. In what follows, we will use the terms RESTful web API, web API, or simply API interchangeably.

Web services often impose dependency constraints that restrict the way in which two or more input parameters can be combined to form valid calls to the service, we call these *inter-parameter dependencies* (or simply *dependencies* henceforth). For instance, it is common that the inclusion of a parameter requires or excludes—and therefore depends on—the use of some other parameter or group of parameters. As an example, the documentation of the Twilio API [47] states that, when sending an SMS, either the `body` parameter or the `media_url` parameter must be set, but not both at the same time. Similarly, the documentation of the QuickBooks payments API [37] explains that, when creating a credit card, at least one of the parameters `region` or `postalCode` must be provided, although both of them are declared as optional.

Current specification languages for RESTful web APIs such as OAS and RAML provide little or no support at all for describing dependencies among input parameters. Instead, they just encourage to describe such dependencies as a part of the description of the parameters in natural language, which may result in ambiguous or incomplete descriptions. For example, the Swagger documentation states[1] "*OpenAPI 3.0 does not support parameter dependencies and*

- *A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés are with the Smart Computer systems Research and Engineering Lab (SCORE) and the Research Institute of Informatics Engineering (I3US), Universidad de Sevilla, Spain. E-mail: {alberto.martin, sergiosegura, aruiz}@us.es*
- *C. Müller is with the Research Institute of Informatics Engineering (I3US), Universidad de Sevilla, Spain. E-mail: cmuller@us.es*

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

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". The lack of support for dependencies means a strong limitation for current specification languages, since without a formal description of such constraints is hardly possible to interact with the services without human intervention. For example, it would be extremely difficult to automatically generate test cases for the APIs of Twilio or QuickBooks without an explicit and machine-readable definition of the dependencies mentioned above. The interest of industry in having support for these types of dependencies is reflected in an open feature request in OAS entitled "Support interdependencies between query parameters", created in January 2015 with the message shown below. At the time of writing this paper, the request has received over 340 votes, and it has received 58 comments from 36 participants [30].

> *"It would be great to be able to specify interdependencies between query parameters. In my app, some query parameters become "required" only when some other query parameter is present. And when conditionally required parameters are missing when the conditions are met, the API fails. Of course I can have the API reply back that some required parameter is missing, but it would be great to have that built into Swagger."*

This feature request has fostered an interesting discussion where the participants have proposed different ways of extending OAS to support dependencies among input parameters. However, each approach aims to address a particular type of dependency and thus show a very limited scope. Addressing the problem of modelling and validating input constraints in web APIs should necessarily start by understanding how dependencies emerge in practice. Inspired by this idea, in a previous paper we conducted a thorough study on the presence of inter-parameter dependencies in industrial web APIs [25]. For that purpose, we reviewed more than 2.5K operations from 40 real-world RESTful APIs from different application domains. As expected, we found that input dependencies are the norm, rather than the exception, with 85% of the reviewed APIs having some kind of dependency among their input parameters. More importantly, as the main outcome of our study, we presented a catalogue of seven types of dependencies consistently found in RESTful web APIs. These findings, and specifically the catalogue of dependencies (described in Section 2), serve as the starting point for this work.

In this article, we first present a domain-specific language for the specification of inter-parameter dependencies in web APIs called *Inter-parameter Dependency Language* (IDL). Second, we present an approach for the automated analysis of IDL specifications using constraint programming. In particular, we present a general-purpose mapping showing how to translate an IDL specification into a constraint satisfaction problem (CSP). Then, we present a catalogue of seven analysis operations of IDL specifications and show how they can be automated using standard constraint programming reasoning operations. For example, given an IDL specification one may be interested to know if it includes any inconsistencies like parameters that cannot be selected (*dead* parameters), i.e., any request including them would violate some dependency, or whether a given call to the API satisfies all the dependencies. Our approach is supported by several tools including an (Eclipse) editor, a parser, an OAS extension (called *IDL4OAS*), a constraint-programming aided library supporting the automated analysis of IDL specifications, and a complete test suite.

To illustrate the potential of the approach, we assess the impact of our contributions in the domain of automated testing of RESTful web APIs. Specifically, 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 three commercial APIs. As expected, random testing struggled to generate valid requests: about 99% of the random requests generated violated inter-parameter dependencies in the APIs of Stripe and YouTube. In contrast, IDLReasoner generated 100% valid requests for all the services under test. More importantly, test cases generated with IDLReasoner revealed 17 times more failures than random testing (1,209 vs. 67).

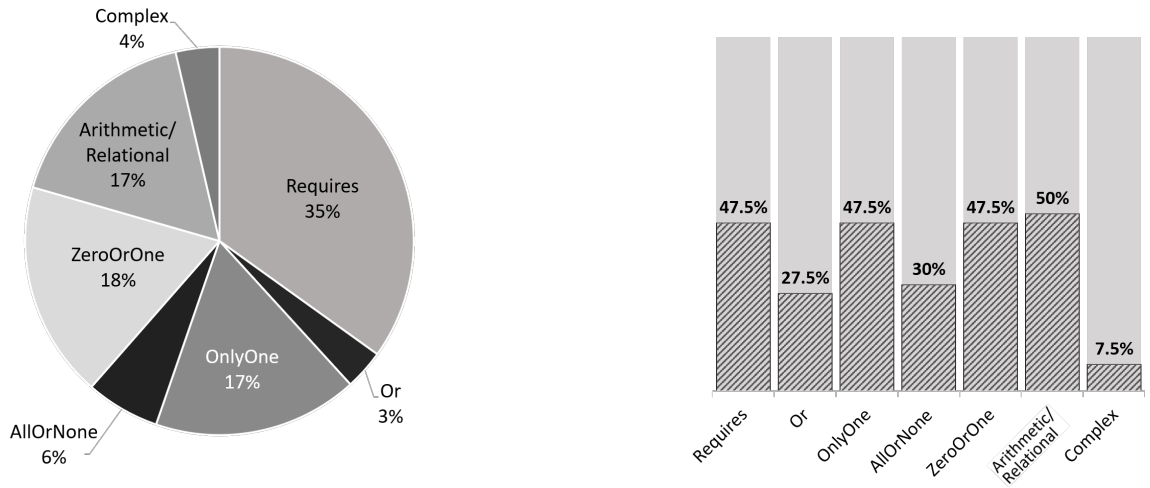
Beyond testing, our contributions prepare the ground for a new range of specification-driven applications in web APIs. For example, an API gateway supporting the automated analysis of IDL could automatically reject requests violating any dependencies, without even redirecting the call to the corresponding service, saving time and user quota. Code generators could automatically include built-in assertions to deal with invalid input combinations, preventing input-validation failures caused by violated dependencies. Analogously, interactive API documentations could be enriched with analysis capabilities to detect invalid calls even before invoking the API. The range of new applications is promising.

This paper is structured as follows: Section 2 presents the catalogue of dependency patterns found in our systematic review of real-world APIs. Section 3 introduces the syntax of IDL using examples. Section 4 describes our approach for the automated analysis of IDL specifications. Our tool suite is presented in Section 5. Section 6 describes the evaluation of our approach. Section 7 describes the possible threats to validity and how these were mitigated. The related work is discussed in Section 8. Finally, Section 9 concludes the paper and presents future lines of research.

2 Catalogue of Dependencies

The contributions presented in this paper are built on the findings of a previous study by the authors on the presence of inter-parameter dependencies in industrial RESTful web APIs [25]. For the sake of understandability and to make our paper self-contained, we next summarise those results more relevant for this article, and redirect the interested reader to the original paper for further details.

In our previous study, we reviewed more than 2.5K operations from 40 real-world RESTful APIs including popular APIs such as those of YouTube, Google Maps, Amazon S3, and PayPal. The results of the study showed that dependencies are extremely common and pervasive—they appear in 85% of the APIs under study (34 out of 40) across all application domains and types of operations. Specifically, we identified 633 dependencies among input parameters



(a) Dependency types.

(b) Occurrences in APIs

Figure 1: Distribution of dependencies by type and percentage of APIs.

in 9.7% of the API operations analysed (248 out of 2,557). The collected data helped us characterise dependencies identifying their most common shape—dependencies in read operations involving two query parameters—, but also exceptional cases such as dependencies involving up to 10 parameters and dependencies among different types of parameters, e.g., header and body parameters. More importantly, we classified the inter-parameter dependencies identified into seven general types, described below.

Before going in depth into each type of dependency, a number of considerations must be taken into account. First, for the sake of simplicity, dependencies are described using single parameters. However, all dependencies can be generalised to consider groups of parameters using conjunctive and disjunctive connectors. Second, dependencies can affect not only the presence or absence of parameters, but also the values that they can take. In what follows, when making reference to a parameter *being present* or *being absent*, it could also mean a parameter *taking a certain value*. Finally, when introducing each dependency type, we will make reference to Figure 1, which shows the distribution of dependencies by type (Figure 1a) and the percentage of subject APIs including occurrences of each dependency type (Figure 1b). Next, we describe the seven types of dependencies found in our study, including examples.

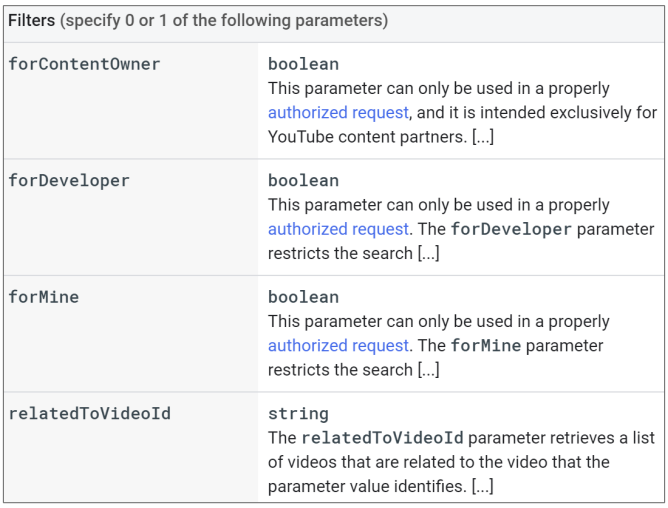
Requires. The presence of a parameter p_1 in an API call requires the presence of another parameter p_2. As previously mentioned, p_1 and p_2 can be generalised to groups of parameters and parameter-value relations. Based on our results, this is the most common type of dependency in web APIs, representing 35% of all the dependencies identified in our study (Figure 1a), and being present in 47.5% of the subject APIs (Figure 1b). As an example, in the GitHub API [17], 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. As illustrated in Figure 1, this type of dependency represents only 3% of the dependencies identified in the subject APIs. Interestingly, however, we found that more than one fourth of the APIs (27.5%) included some occurrence of this dependency type, which suggests that its use is fairly common in practice. As an example, in the Google Maps Places API [18], when searching for places, both `query` and `type` parameters are optional, but at least one of them must be used.

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. As observed in Figure 1, this group of dependencies represent 17% of all the dependencies identified, and they appear in almost half of the APIs under study (47.5%). Among others, we found that this type of dependency is very common in APIs from the category *media*, where a resource can be identified in multiple ways, e.g., a song can be identified by its name or by its ID, and only one value must be typically provided. For example, in the Last.fm API [23], 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. Very similarly to the *Or* dependency type, only 6% of the dependencies found belong to this category, nonetheless, they are present in about one third of the APIs under study (30%). In the Yelp API [53], 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. Figure 1 reveals that this dependency type is common both in terms of the number of occurrences (18% of the total) and the number of APIs including it (47.5%). As an example, in the search operation of the YouTube API [55], it is possible to filter results



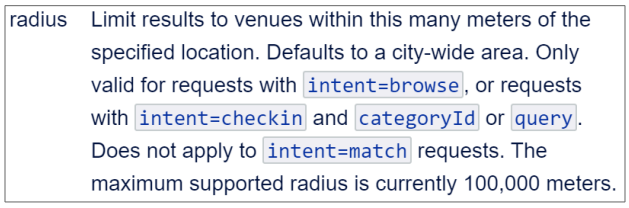
Filters (specify 0 or 1 of the following parameters)	
forContentOwner	boolean This parameter can only be used in a properly authorized request, and it is intended exclusively for YouTube content partners. [...]
forDeveloper	boolean This parameter can only be used in a properly authorized request. The forDeveloper parameter restricts the search [...]
forMine	boolean This parameter can only be used in a properly authorized request. The forMine parameter restricts the search [...]
relatedToVideoId	string The relatedToVideoId parameter retrieves a list of videos that are related to the video that the parameter value identifies. [...]

Figure 2: *ZeroOrOne* dependency in the YouTube API.

with four optional but mutually exclusive parameters, as depicted in Figure 2.

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., $p_1 + p_2 < 100$. As shown in Figure 1, this type of dependency is the most recurrent across the subject APIs, being present in half of them. Moreover, 17% of the dependencies found are of this type. An example of a relational dependency is found in the Twitter API [54]: when searching for tweets, the `max_id` parameter must be greater than or equal to the `since_id` parameter, otherwise no tweets will be returned. In the payments API Forte [11], the following arithmetic dependency exists: when creating a merchant application, this can be owned by up to four businesses, in which case the sum of the percentages cannot be greater than 100.

Complex. These dependencies involve two or more of the types of constraints previously presented. Based on our results, they are typically formed by a combination of *Requires* and *OnlyOne* dependencies. As illustrated in Figure 1, we found 4% of complex dependencies, being present in 7.5% of the subject APIs. Figure 3 depicts a *Complex* dependency present in the Foursquare API [12]: if `radius` is used, then either `intent` is set to `'browse'` or `intent` is set to `'checkin'` and `categoryId` or `query` are present too.



radius	Limit results to venues within this many meters of the specified location. Defaults to a city-wide area. Only valid for requests with `intent=browse`, or requests with `intent=checkin` and `categoryId` or `query`. Does not apply to `intent=match` requests. The maximum supported radius is currently 100,000 meters.

Figure 3: *Complex* dependency present in the `GET /venues/search` operation of the Foursquare API.

3 Inter-parameter Dependency Language

In this section, we present *Inter-parameter Dependency Language* (IDL), a textual domain-specific language for the specification of dependencies among input parameters in web APIs. Specifically, IDL is designed to express the seven types of inter-parameter dependencies identified in our study on real-world APIs and described in the previous section. This includes support for the dependencies discussed in the related OAS feature request in GitHub [30]. For the design of the language, we took inspiration from the input format of the combinatorial testing tool Pairwise Independent Combinatorial Testing (PICT) [35], by Microsoft, where constraints among input parameters can be defined using invariants, conditional definitions (if/then/else), logical operators and relational operators.

It is worth mentioning that IDL focuses on the definition of dependencies among parameters, but not in the definition of the parameters themselves. This is because IDL is specifically designed to be easily integrated into API specification languages such as OAS or RAML, where parameters are specified in different ways. Thus, in what follows, we simply assume that each parameter has a name and a domain.

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 [19] and on the supplemental material provided with this article [43].

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's 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. This relation can be defined using standard relational operators (e.g., `p1>=100`) or a wild card match—using the operator LIKE—if the parameter is a string (`PATTERN_STRING` in Listing 1), with '*' meaning zero or more characters and '?' meaning one character (e.g., `p3 LIKE 'test_*'`). 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 priority. In what follows, we describe the IDL notation of each type of dependency.

Requires. This type of dependency is expressed as "`IF predicate THEN predicate;`", where the first predicate is the *condition* and the second is the *consequence*. The following listing shows two examples. Dependency in line 2, for instance, indicates that invocations including the parameters `p1` and `p2` should not include `p3` nor `p4`, otherwise the call would be invalid.

```
1 IF p1 THEN p2=='A';
2 IF p1 AND p2 THEN NOT (p3 OR p4);
```

Or. This type of dependency is expressed using the keyword "Or" followed by a list of two or more predicates placed inside parentheses: "`Or(predicate, predicate [, ...]);`". The dependency is satisfied if at least one of the predicates evaluates to true. Two examples follow. Dependency in line 1, for instance, specifies that valid invocations should include at least one of the parameters `p1`, `p2` or `p3`.

```
1 Or(p1, p2, p3);
```

```
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. `STRING`, `DOUBLE` and `BOOLEAN` are standard data types.

```
2 Or(p1, p3 AND p5, p6=='B');
```

OnlyOne. These dependencies are specified using the keyword "OnlyOne" followed by a list of two or more predicates placed inside parentheses: "`OnlyOne(predicate, predicate [, ...]);`". The dependency is satisfied if one, and only one of the predicates evaluates to true. Examples of this dependency are shown below. The dependency in line 1, for example, indicates that valid invocations should include either the parameter `p1` or the parameter `p2` with value `'B'`, but not both at the same time.

```
1 OnlyOne(p1, p2=='B');
2 OnlyOne(p1 OR p2, p3 AND (p4 OR p5));
```

AllOrNone. This type of dependency is specified using the keyword "AllOrNone" followed by a list of two or more predicates placed inside parentheses: "`AllOrNone(predicate, predicate [, ...]);`". The dependency is satisfied if either all the predicates evaluate to true, or all of them evaluate to false. The dependency in line 1 below, for instance, indicates that valid calls are those including either the parameter `p1` and `p2` with value `true`, or conversely, those not including the parameter `p1` and not including `p2` with value `true`.

```
1 AllOrNone(p1, p2==true);
2 AllOrNone(p1 AND p2, p3 LIKE 'test_*' OR p4<10);
```

ZeroOrOne. These dependencies are specified using the keyword "ZeroOrOne" followed by a list of two or more predicates placed inside parentheses: "`ZeroOrOne(predicate, predicate [, ...]);`". The dependency is satisfied if none or at most one of the predicates evaluates to true. Two examples follow. Line 2, for instance, specifies that valid invocations must meet zero or one (but not both) of the two conditions between parentheses: (1) including the parameter `p1`, or (2) including the parameter `p2` with a value less than or equal to 100.

```
1 ZeroOrOne(p1, p2, p3, p4);
2 ZeroOrOne(p1, p2<=100);
```

Arithmetic/Relational. Relational dependencies are specified as pairs of parameters joined by any of the following relational operators: ==, !=, <=, <, >= or > (see examples in lines 1 and 2 below). Arithmetic dependencies relate two or more parameters using the operators +, - , *, / followed by a final comparison using a relational operator. Lines 3 and 4 of the following listing show examples of arithmetic dependencies.

```
1 p1 < p2;
2 p1 != p2;
3 p1 + p2 - p3 * p4 == 100;
4 p1 * p2 / ((p3 - p4) * p5) < 176.89;
```

Complex. These dependencies are specified as a combination of the previous ones since, as previously mentioned, predicates can contain other dependencies. As an exception to this rule, predicates cannot include *Requires* dependencies to avoid over-complicated specifications (such dependencies can be expressed in simpler ways). The following listing shows some examples of complex dependencies. Dependency in line 1 combines four different types of dependencies: *Requires*, *ZeroOrOne*, *OnlyOne* and *Relational*.

```
1 IF p1 THEN ZeroOrOne(p2, OnlyOne(p3, p4>p5));
2 AllOrNone(p1+p2<100, Or(p3=='A', Or(p4, p5>p6)));
```

It is worth making a few general clarifications about the language regarding dependencies *Or*, *OnlyOne*, *AllOrNone* and *ZeroOrOne*. These are not strictly necessary, as they could be translated to several *Requires* dependencies. However, they are provided as syntactic sugar to make specifications succinct and self-explanatory. An example is given in the following IDL excerpt (lines 1-3). Secondly, they cannot contain negated elements within their parentheses, since such constraints can be expressed in simpler ways (lines 5-6). Finally, they can optionally be preceded by the keyword "NOT" to negate the meaning of the constraint (see line 8 below for an example).

```
1 AllOrNone(p1, p2);    // Equivalent to 1) and 2):
2 IF p1 THEN p2;        // 1)
3 IF p2 THEN p1;        // 2)
4
5 Or(p1, NOT p2);       // Invalid dependency
6 IF p2 THEN p1;        // Equivalent to line 5
7
8 NOT OnlyOne(p1, p2); // Valid negated dependency
```

Listing 2 depicts the IDL specification of the Google Maps Places API [18]. It comprises seven operations, four of which have dependencies. The API has eight dependencies in total, including six out of the seven types of dependencies supported in IDL (all of them except the complex ones), namely:

- Line 2: If the parameter `radius` is used, then `rankby` cannot be set to `'distance'`, and vice versa.
- Line 3: If the parameter `rankby` is set to `'distance'`, then at least one of the following parameters must be present: `keyword`, `name` or `type`.
- Line 4: The parameter `maxprice` must be greater than or equal to `minprice`.
- Line 7: Either both `location` and `radius` are used, or none of them.
- Line 8: `query` and `type` are both optional parameters, but at least one of them must be used.
- Line 9: Equal to line 4.
- Line 12: One, and only one of the parameters `maxheight` and `maxwidth` must be used.
- Line 15: If the parameter `strictbounds` is used, then both `location` and `radius` must be used too.

```
1  // Operation: Search for places within specified area:
2  ZeroOrOne(radius, rankby=='distance');
3  IF rankby=='distance' THEN keyword OR name OR type;
4  maxprice >= minprice;
5
6  // Operation: Query information about places:
7  AllOrNone(location, radius);
8  Or(query, type);
9  maxprice >= minprice;
10
11 // Operation: Get photo of place:
12 OnlyOne(maxheight, maxwidth);
13
14 // Operation: Automcomplete place name:
15 IF strictbounds THEN location AND radius;
```

Listing 2: IDL specification of Google Maps Places API.

4 Automated Analysis

The analysis of IDL deals with the extraction of information from IDL specifications. For example, given an IDL specification, we might be interested to know whether it contains errors (e.g., inconsistent dependencies) or whether a given API call is valid, i.e., it meets all the constraints defined in the specification. Performing these analyses manually is hardly possible in practice.

In what follows, we present our approach for the automated analysis of IDL specifications using constraint programming. In particular, we first present the formal semantics of IDL by explaining how IDL specifications can be mapped to a constraint satisfaction problem (CSP). Then, we present a catalogue of seven analysis operations of IDL specifications and show how they can be automated using standard constraint programming reasoning operations.

4.1 Formal Semantics of IDL

The primary objective of formalising IDL is to establish a sound basis for the automated support. Following the formalisation principles defined by Hofstede et al. [45], we follow a transformational style by translating IDL specifications to a target domain suitable for the automated analysis (*Primary Goal Principle*). Specifically, we propose translating IDL specifications to a CSP that can be then analysed using state-of-the-art constraint programming tools. A similar approach was followed by the authors to automate the analysis of feature models [3] and service level agreements [27], [28].

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.

Table 1 describes the mapping from IDL to CSP. The first row of the table depicts how each input parameter is mapped to CSP variables, domains and constraints. Recall that both the name and domain of each parameter should be taken from the API specification (c.f. p_i and $domain()$ function). For each parameter, two CSP variables are created: (1) one representing the parameter itself (c.f. p_i), and (2) a Boolean variable to express whether the parameter is set or not (c.f. p_iSet). Optionally, we may also get information from the specification about whether each parameter is required (mandatory) or not. If a parameter p_i is required (i.e., it must be present in all API calls), the constraint $p_iSet == true$ is added to the set of constraints C. The second and third rows of the mapping in Table 1 express how the terms are mapped to a CSP. Every time a parameter is found in a predicate, it must be checked whether the parameter is present in the API request. If so, it will evaluate to true, otherwise it will evaluate to false (c.f. $p_iSet == true$ from the second row of the table). In the case of parameters having a relational condition with a value, it must also be checked that the parameter satisfies such condition (c.f. third row of the table). Finally, predicates and dependencies are defined recursively using the function $map(E)$, where E is either a term, a predicate or a dependency. Exceptionally, relational and arithmetic dependencies are only evaluated if all the involved parameters are present in the API request (c.f. last two rows in Table 1).

As an example, Listing 3 shows the resulting CSP obtained as a result of applying the proposed mapping to the IDL specification of the *Search* operation in the Google Maps Places API, specified in Listing 2 (lines 1-4). Analogously, Listings 4 and 5 depict the CSP constraints derived from the *Query* and *Get* operations in Listing 2, respectively (lines 6-9 and 11-12).

```
1  V = { radius, radiusSet, rankby, rankbySet, keyword,
2        keywordSet, name, nameSet, type, typeSet,
3        maxprice, maxpriceSet, minprice, minpriceSet    }
4
5  D = { int, Boolean, string, Boolean, string, Boolean,
6        string, Boolean, string,  Boolean, int, Boolean,
7        int, Boolean                                    }
8
9  C = {//ZeroOrOne(radius, rankby=='distance');
10     ((radiusSet==true  ⟹ ¬(rankbySet==true AND
11     rankby=='distance') AND ((rankbySet==true AND
12     rankby=='distance')  ⟹ ¬radiusSet==true)) OR
13     ((¬radiusSet==true) AND ¬(rankbySet==true
14     AND rankby=='distance'))) AND
15     //IF rankby=='distance' THEN keyword OR name OR
16     //   type;
17     ((rankbySet==true AND rankby=='distance') ⟹
18     ((keywordSet==true) OR (nameSet==true) OR
19     (typeSet==true))) AND
20     //maxprice >= minprice;
21     (((maxpriceSet==true) AND (minpriceSet==true))  ⟹
22     (maxprice ≥ minprice))                          }
```

Listing 3: CSP of *Search* operation in Listing 2.

```
1  C = {//AllOrNone(location, radius);
2     ((locationSet==true  ⟹  radiusSet==true) AND
3     (radiusSet==true  ⟹  locationSet==true) AND
4     (¬locationSet==true  ⟹ ¬radiusSet==true) AND
5     (¬radiusSet==true ⟹ ¬locationSet==true)) AND
```

Mapping from IDL to CSP		
API Parameters		CSP Mapping
	[Parameters] P	$\forall p_i \in P, \begin{cases} V \leftarrow V \cup p_i \cup p_i Set \\ D \leftarrow D \cup domain(p_i) \cup Boolean \\ C \leftarrow C \cup p_i Set == true \text{ (if } p_i \text{ is required)} \end{cases}$
IDL Element		CSP Mapping
Terms: map(T)	[Parameter] p_i	$C \leftarrow C \cup \{p_i Set == true\}$
	[Parameter-Value Relation] $p_i\ relOp^{\star}\ v$	$C \leftarrow C \cup \{p_i\ relOp\ v \wedge\ p_i Set == true\}$
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] $p_i\ relOp^{\star}\ p_j$	$C \leftarrow C \cup \{(p_i Set == true \wedge p_j Set == true) \implies p_i\ relOp\ p_j\}$
	[Arithmetic Dependency] $p_i\ arOp^{\diamond}\ p_j\ arOp ...\ p_n\ relOp^{\star}\ v$	$C \leftarrow C \cup \{(p_i Set == true \wedge p_j Set == true \wedge\ ...\ p_n Set == true) \implies (p_i\ arOp\ p_j\ arOp\ ...\ p_n\ relOp\ v)\}$

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

Table 1: IDL to CSP mapping.

```
    //Or(query, type);
    (querySet==true OR typeSet==true) AND
    //maxprice >= minprice;
    (((maxpriceSet==true) AND (minpriceSet==true)) ⟹
    (maxprice ≥ minprice))                          }
```

Listing 4: Constraints of *Query* operation in Listing 2.

```
C = {//OnlyOne(maxheight, maxwidth);
   ((maxheightSet==true ⟹ ¬maxwidthSet==true) AND
   (maxwidthSet==true ⟹ ¬maxheightSet==true))      }
```

Listing 5: Constraints of *Get* operation in Listing 2.

4.2 Analysis Operations

In this section, we propose a catalogue of seven analysis operations on IDL specifications. These operations leverage the formal description of the dependencies using IDL to extract helpful information such as identifying inconsistencies or checking whether an API call is valid or not. Analogous analysis operations have been defined in the context of the automated analysis of feature models [3] and service level agreements [27], [28]. We may remark that it is not our intention to propose an exhaustive set of analysis operations as that would exceed the scope of this article.

For the description of the operations in CSP, we will refer to the input IDL specification of an API operation `IDL` and the list of the parameters of the API operation `P`, taken from the API specification. Note that `P` is necessary since it contains the information about the types of the parameters, and whether each parameter is defined as required or optional in the API specification. Additionally, we will use the following auxiliary operations:

- `map(IDL,P)`. This operation translates an input IDL specification `IDL` and the list of parameters `P` from the API specification to a CSP following the mapping described in Section 4.1.
- `solve(CSP)`. This standard CSP-based operation returns a random solution for the input `CSP` (if any).
- `solveAll(CSP)`. This standard CSP-based operation returns all the solutions of the input `CSP` (if any).
- `filter(CSP,L)`. This operation takes as input a `CSP` and a list `L` of pairs variable-value to be set, $\{\{p_1, v_1\}, \{p_2, v_2\}, \ldots, \{p_n, v_n\}\}$, and returns the input `CSP` with additional constraints setting each variable in `L`, p_i, to its corresponding value v_i, i.e., $C \leftarrow C \cup \{p_i = v_i\}$.

In what follows, for each operation, we provide a name, a description, an example, and an explanation of how it is mapped to a CSP.

Consistent specification. This operation receives as input the IDL specification of an API operation and its list of parameters, and returns a Boolean indicating whether the specification is consistent or not. An IDL specification is *consistent* if there exists at least one request satisfying all the dependencies of the specification. Inconsistent specifications are the result of users' mistakes and therefore automating their detection can be very helpful. This operation can be translated to a CSP as follows:

$$\text{isConsistentIDL}(\text{IDL}, \text{P}) \iff \text{solve}(\text{map}(\text{IDL}, \text{P})) \neq \emptyset$$

Dead parameter. This operation receives as input the IDL specification of an API operation, its list of parameters, and the name of a parameter, and it returns a Boolean indicating whether the parameter is dead or not. A parameter is *dead* if it cannot be included in any valid call to the service. Dead parameters are caused by inconsistencies in the specification or the design of the service. They may be hard to detect when the inconsistency is caused by several inter-related dependencies. For example, in the following IDL specification, the parameter `p1` is dead since both constraints cannot be satisfied at the same time.

```
IF p1 THEN p2;
OnlyOne(p1, p2);
```

Given an input parameter `p`, this operation can be automated by setting the CSP variable representing the presence of `p` to true ($pSet = true$) and checking whether the problem has at least one solution. If there is no solution, it means that `p` is dead, namely:

$$\text{isDeadParameter}(\text{IDL}, \text{P}, \text{p}) \iff \text{solve}(\text{filter}(\text{map}(\text{IDL}, \text{P}), \{\{\text{pSet}, \text{true}\}\})) = \emptyset$$

False optional. This operation assumes that the specification of each parameter indicates, as in OAS, whether the parameter is required (i.e., it must be included in every service request) or optional. This operation takes as input the IDL specification of an API operation, its list of parameters, and the name of a parameter specified as optional, and returns a Boolean indicating whether the parameter is false optional or not. A parameter is *false optional* if it is required (i.e., it must be included in all API calls to satisfy inter-parameter dependencies) despite being defined as optional. False optional parameters should be avoided since they give the user a wrong idea of the domain. For example, suppose that the parameter `p1` is defined as mandatory (e.g., "`required`": `true` in OAS) and `p2` is declared as optional ("`required`": `false`). The constraint "`IF p1 THEN p2`" in IDL would make `p2` a false optional parameter.

Given an input parameter specified as optional `p`, this operation can be automated setting the CSP variable representing the presence of `p` to false ($pSet = false$) and checking whether the problem has at least one solution. If it has no solutions, `p` is false optional. Note that the input IDL specification should be consistent, otherwise all parameters would be classified as false optional. This operation can be translated to a CSP as follows:

$$\text{isFalseOptional}(\text{IDL}, \text{P}, \text{p}) \iff \text{isConsistentIDL}(\text{IDL}, \text{P}) \wedge \text{solve}(\text{filter}(\text{map}(\text{IDL}, \text{P}), \{\{\text{pSet}, \text{false}\}\})) = \emptyset$$

Valid specification. This operation receives as input the IDL specification of an API operation and its list of parameters, and returns a Boolean indicating whether the specification is valid or not. An IDL specification is *valid* if it is consistent (i.e., there exists at least one request satisfying all the dependencies of the specification) and it does not contain any dead or false optional parameters. This operation, defined as a composition of the previous ones, can be helpful to easily detect errors when editing service specifications. This operation can be translated to a CSP as follows:

$$\text{isValidIDL}(\text{IDL}, \text{P}) \iff \text{isConsistentIDL}(\text{IDL}, \text{P}) \wedge \forall \text{p}_\text{i} \in \text{P}(\neg\text{isDeadParameter}(\text{IDL}, \text{P}, \text{p}_\text{i}) \wedge \neg\text{isFalseOptional}(\text{IDL}, \text{P}, \text{p}_\text{i}))$$

Valid request. This operation takes as input the IDL specification of an API operation, its list of parameters, and a service request (i.e., a list of parameters and their values) and returns a Boolean indicating whether the request is valid or not. A service request is valid if it satisfies all the dependencies of the IDL specification. For example, the following is a valid request for the IDL specification depicted in Listing 6: `{p1=2,p2=5}`.

```
Or(p1, p2 AND p3);
OnlyOne(p2, p3);
```

Listing 6: Valid IDL specification.

Let `R` be an input request, i.e., a list of parameters and their respective values. This operation can be translated to a CSP by (1) setting the CSP variables related to each parameter to the value indicated in R, (2) setting the CSP variables related to the presence of the parameters in R to true ($R_i Set = true$), (3) setting the CSP variables related to the parameters not included in R to false ($O_i Set = false$ where $O = P \setminus R$), and (4) checking whether the problem has at least one solution. If it has no solutions, it means that the request is not valid, namely:

$$\text{isValidRequest}(\text{IDL}, \text{P}, \text{R}) \iff \text{O} = \text{P} \setminus \text{R} \wedge \text{solve}(\text{filter}(\text{map}(\text{IDL}, \text{P}), \text{R} \cup \{\{\text{R}_1\text{Set}, \text{true}\}, \{\text{R}_2\text{Set}, \text{true}\}, \ldots, \{\text{R}_\text{n}\text{Set}, \text{true}\} \{\text{O}_1\text{Set}, \text{false}\}, \{\text{O}_2\text{Set}, \text{false}\}, \ldots, \{\text{O}_\text{k}\text{Set}, \text{false}\}\})) \neq \emptyset$$

Valid partial request. This operation is analogous to the previous one but the input request is *partial* or incomplete, meaning that some other parameters should still be included to make it a full valid request. This operation returns a Boolean indicating whether the partial request is valid. A partial request is valid if it does not include any contradiction, i.e., it can be extended with new parameters to become a valid request.

Let S be a partial input request. This operation can be specified as a CSP as follows:

$$\begin{aligned}&\text{isValidPartialRequest}(\text{IDL}, \text{P}, \text{S}) \iff \\ &\quad \text{solve}(\text{filter}(\text{map}(\text{IDL}, \text{P}), \text{S} \cup \\ &\quad \{\{\text{S}_1\text{Set}, \text{true}\}, \{\text{S}_2\text{Set}, \text{true}\}, \ldots, \{\text{S}_\text{n}\text{Set}, \text{true}\})) \neq \emptyset\end{aligned}$$

Random request. This operation receives as input the IDL specification of an API operation and its list of parameters, and returns a random valid request for the operation. This operation can be automated by translating the IDL specification to a CSP and requesting the solver to find a random solution, namely:

$$\text{randomRequest}(\text{IDL}, \text{P}) = \text{solve}(\text{map}(\text{IDL}, \text{P}))$$

Table 2 summarizes some of the potential applications of the proposed analysis operations. As illustrated, four out of the seven operations are intended to automatically identify errors in IDL specifications. Note that although the operations *Consistent specification*, *Dead parameter* and *False optional* are related, each of them addresses a specific type of inconsistency and therefore they are helpful to point the users towards the specific types of errors found in their specification. The operation *Valid request* has applications in testing and monitoring. For example, complex test data generators could use this operation to check whether the generated API requests are valid or not. Also, an API gateway supporting this operation could detect and monitor invalid calls without the need to redirect the request to the target service, providing faster responses and reducing the consumption of user quota. The operation *Valid partial request* may be helpful for the early detection of inconsistencies. For example, an interactive API documentation supporting this operation could warn the user about inconsistencies as soon as a dependency is violated, without having to wait until constructing the full request. Finally, the operation *Random request*, in combination with test data generators, can be very useful for testing of web services as shown in Section 6.

Operation	Applications
Consistent specification	Specification error detection
Dead parameter	Specification error detection
False optional	Specification error detection
Valid specification	Specification error detection
Valid request	Testing, monitoring
Valid partial request	Interactive documentation
Random request	Testing

Table 2: Potential applications of the analysis operations.

5 Tooling Support

As a part of our contribution, we provide a set of tools supporting the specification and analysis of inter-parameter dependencies in web APIs, including an editor of IDL specifications, an extension for the OAS language and an analysis library supporting the integration of our approach into any external project. Together, these components make our work readily applicable in practice and provide a reference implementation for future contributions on the topic.

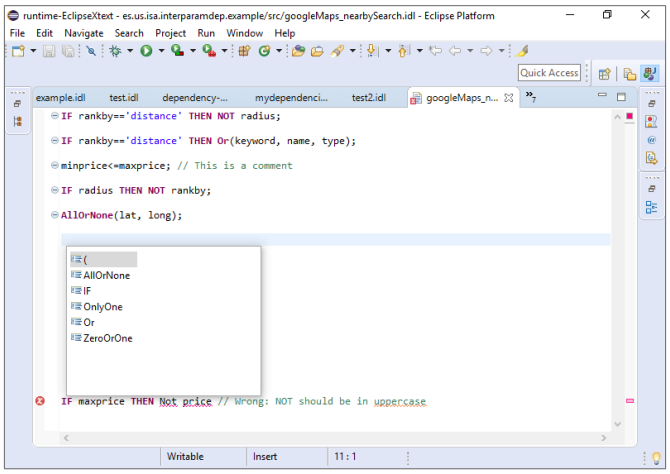


Figure 4: IDL Editor.

5.1 IDL Editor and Parser

We implemented IDL using Xtext [50], a popular framework for the development of programming languages and DSLs. Xtext takes a grammar as input and generates a complete set of tools as output, including a linker, a compiler, a parser and a fully-fledged editor supporting features such as code completion, type checking, syntax coloring and validation. A simplified version of the IDL grammar is provided in Listing 1, the full version is available as a part of the implementation of IDL [19] (and also on the supplemental material provided with this article [43]). Figure 4 depicts a screenshot of the editor, showing some of its capabilities: code completion, syntax coloring and error checking. The editor is based on Eclipse, but is compatible with any web browser or IDE supporting the Language Server Protocol [24].

5.2 IDL4OAS: An OAS Extension

In order to foster the adoption of our approach, we propose an extension of OAS for the specification of inter-parameter dependencies using IDL. We call this extension IDL4OAS. An OAS document describes a REST API in terms of the elements it comprises, namely paths, operations, resources, request parameters and responses. OpenAPI provides a way to add extra information that may not be supported natively. This information is included in custom properties that start with "x-", called *extensions*. IDL4OAS is an OAS extension that allows to specify a set of IDL dependencies for each API operation. An extra property called "x-dependencies" must be added at the operation level, including the set of dependencies among the input parameters of the operation. Listing 7 shows an excerpt of an OAS document extended with IDL4OAS, corresponding to the *Search* operation from the Google Maps Places API (see Listing 2).

As illustrated, the property "x-dependencies" has been added to the "GET /search" operation. This property is actually an array of elements, where each element represents a single dependency, therefore they must be preceded by hyphens, following the YAML syntax.

```
paths:
  /search:
    get:
      parameters:
      - name: radius [...]
      - name: rankby [...]
      - name: keyword [...]
      - name: name [...]
      - name: type [...]
      - name: minprice [...]
      - name: maxprice [...]
      - [...]
      [...]
      x-dependencies:
      - ZeroOrOne(radius, rankby=='distance');
      - IF rankby=='distance' THEN keyword OR name OR
          type;
      - maxprice >= minprice;
```

Listing 7: OAS document of the search operation from the Google Maps Places API extended with IDL4OAS.

5.3 IDLReasoner: An Analysis Library

In this section, we present IDLReasoner, a CSP-based Java library that allows to programmatically analyse IDL documents. Specifically, IDLReasoner translates input IDL specifications to CSPs using MiniZinc [26], a constraint solving language designed for modelling optimisation problems in a high-level, solver-independent way. This allows IDLReasoner to be used with any CSP solver supporting MiniZinc as an input format.

Figure 5 shows the high-level architecture of IDLReasoner, using a UML component diagram. The library comprises three main components: the *Mapper*, which translates variables from the API specification and dependencies from IDL to MiniZinc, and manipulates the resulting MiniZinc file accordingly for each analysis operation; the *Resolutor*, which performs the calls to the selected CSP solver; and the *Analyzer*, which leverages the Mapper and the Resolutor components to execute the seven analysis operations from the catalogue.

IDLReasoner works as follows: the Analyzer takes three elements as input, namely, an IDL document, an API specification (e.g., OAS) and the API operation where the dependencies are present (e.g., "GET /search"). First, the Mapper transforms the API operation parameters, their domains and the IDL dependencies to a MiniZinc file, representing a CSP. The *ConstraintsMapper* component leverages the IDL parser to translate IDL dependencies into MiniZinc constraints. Then, when an analysis operation is invoked in the Analyzer component (e.g., valid specification), the Mapper manipulates the CSP file accordingly and the Resolutor calls the CSP solver on the manipulated file. IDLReasoner supports the seven analysis operations explained in Section 4.2. It is worth mentioning that IDLReasoner supports both IDL and OAS documents separately, as well as OAS documents including the specification of dependencies with IDL4OAS.

IDLReasoner is developed with extensibility in mind. It can be extended to multiple web API specification languages and CSP solvers. At the time of writing this article, IDLReasoner supports OAS (and IDL4OAS), and a range of CSP solvers compatible with MiniZinc, including Chuffed [7] and Gecode [16].

For the validation of the tool, we developed a test suite of 203 test cases using standard testing techniques including equivalence partitioning, boundary-value analysis,

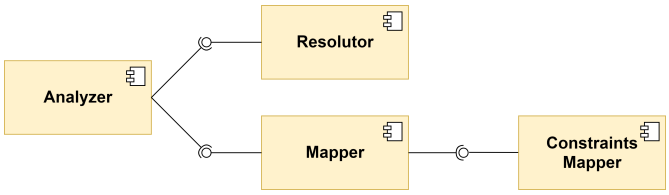


Figure 5: High-level architecture of IDLReasoner.

and combinatorial testing. Among other issues, fully documented in GitHub [19], [20], we detected and fixed faults related to the parsing of IDL specifications, their translation to MiniZinc files and the behaviour of the analysis operations for boundary cases such as operations without parameters. Although performance is out of the scope of our work, it is worth mentioning that the execution of the whole suite took between 130 and 140 seconds in a standard PC running an Intel i5 processor with 16GB of RAM and a solid-state drive (SSD). All the test cases, including their test inputs and expected outputs, are publicly available, as well as their implementation in JUnit [20].

6 Application to Automated Testing

In this section, we report the results of two experiments showing the potential of our approach in the context of automated testing of RESTful web services.

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 the services with random requests conforming to the OpenAPI specification of the API [1]. 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. In what follows, we report the results of two experiments 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, an open-source testing framework for RESTful web APIs developed by the authors and available on GitHub [40]. In particular, we aim to answer the following research questions:

- **RQ1**: *What is the effectiveness of IDLReasoner in generating valid requests for real-world APIs containing inter-parameter dependencies?*
- **RQ2**: *What is the effectiveness of IDLReasoner in detecting failures in real-world APIs containing inter-parameter dependencies?*

To answer these questions, we selected three API operations with inter-parameter dependencies from three industrial APIs used by millions of users. Table 3 provides a summary of the services under test. For each service, the table shows the API name, description of the operation tested, number of input parameters (P), number of IDL dependencies (D), and number (and percentage) of different parameters involved in at least one dependency (PD).

API	Operation	P	D	PD (%)
Stripe	Create product	18	6	11 (61%)
Yelp	Search businesses	14	3	7 (50%)
YouTube	Search	31	16	25 (81%)

Table 3: RESTful API operations used in the evaluation.

It is worth noting that, in the following experiments, we used five out of the seven analysis operations presented in Section 4.2: *Valid specification*, *Consistent specification*, *Dead parameter*, *False optional* and *Random request*. The first four operations were used to identify potential errors in the IDL specifications of the services, manually written by the authors, while the last one was used to generate test cases (i.e., API requests).

6.1 RQ1: Generation of Valid Requests

In this experiment, we compare the effectiveness of a random test case generator and IDLReasoner in generating valid requests, i.e., those satisfying all input constraints, including inter-parameter dependencies.

Setup. For each API operation under test, we generated 1,000 random requests and 1,000 requests with IDLReasoner (c.f. operation *Random request* from Section 4.2). Then, we counted the number of actual valid requests based on the 2XX (successful status codes) responses obtained.

SUT	Random	IDLReasoner
Stripe	1.3%	100%
Yelp	54.6%	97.1%
YouTube	1.6%	100%

Table 4: Percentage of valid requests generated.

Results. Table 4 shows the percentage of valid requests generated by each approach. As expected, the random test case generator was hardly able to generate valid requests for highly constrained APIs like Stripe (1.3%) and YouTube (1.6%). In contrast, IDLReasoner generated 100% of valid requests in all the APIs under test except Yelp, where a few valid requests obtained error responses due to actual faults, e.g., when setting the `open_at` parameter to a very large integer, a 500 status code is returned. Based on these results, we can answer RQ1 as follows: IDLReasoner effectively supports the generation of valid requests for RESTful web services, outperforming current random approaches where inter-parameter dependencies are ignored.

6.2 RQ2: Detection of Failures

In this experiment, we compare the number of failures uncovered by randomly generated requests, and requests generated with IDLReasoner.

Setup. For each API and test generation technique (denoted as "Random" and "IDLReasoner" in Table 5), we generated 2,000 requests, half valid and half invalid. Half of the invalid requests generated with IDLReasoner were constructed by violating one or more dependencies.

API	Random	IDLReasoner
Stripe	0	535
Yelp	67	161
YouTube	0	513
Total	67	1,209

Table 5: Failures classified by test generation technique.

Results. Table 5 depicts the number of failures revealed by each technique. The randomly generated requests revealed 67 failures related to 500 status codes and disconformities with the OAS specification in the Yelp service. IDLReasoner, on the other hand, found more (1,209) and more complex bugs. For example, in the Yelp service, when searching for businesses in Egypt in Finnish language, the error `LOCATION_NOT_FOUND` (400 status code) is returned, but this does not happen for other languages such as Italian, even though the results are in English. Other failures are related to dependencies wrongly specified in the API documentation or badly implemented in the API itself. A description of all the bugs found is available in the supplementary material of the paper [43]. As a result, we can answer RQ2 as follows: IDLReasoner is effective in uncovering failures in real-world APIs, outperforming state-of-the-art random test case generators.

7 Threats to Validity

Next, we discuss the possible internal and external validity threats that may have influenced our work, and how these were mitigated.

7.1 Internal Validity

Threats to the internal validity relate to those factors that might introduce bias and affect the results of our investigation. One of the main threats in this regard is the subjective and manual review process conducted for identifying inter-parameter dependencies in the online documentation of the subject APIs. Some dependencies might have been misclassified or simply overlooked. To mitigate this threat, the documentation of each API was carefully checked several times, recording all the relevant information for its later analysis, and also to enable replicability [9]. The impact of possible mistakes was also minimised by the large number of APIs and operations reviewed (40 APIs and 2,557 operations), which makes us remain confident of the overall accuracy of the results.

Another possible threat is related to the existence of bugs in the implementation of the tools provided. To mitigate

this threat, both the DSL and the analysis library have been thoroughly tested using standard testing techniques such as equivalence partitioning and combinatorial testing. In total, we modelled 267 dependencies in IDL (96 artificial and 171 from real-world APIs), covering the seven types of dependencies identified in our study [25], including multiple and varied combinations of them, and we created 203 JUnit test cases for IDLReasoner. Furthermore, the tools with their test suites and the results of our experiments are freely available [43], thereby allowing full replication of the evaluation performed.

7.2 External Validity

This concerns the extent to which we can generalise from the results of the experiments. Our study on the existence of inter-parameter dependencies in practice is based on a subset of 40 web APIs, and thus our results may not generalise to other APIs. To minimise this threat, we systematically selected a large set of real-world APIs from multiple application domains, including some of the most popular APIs in the world with millions of users worldwide. The same threat may be considered for the evaluation performed, since we only tested three APIs. However, note that the purpose of such evaluation is to demonstrate the potential of the approach proposed in this paper. A more comprehensive evaluation remains to be done in future work.

As another threat, the DSL proposed in this paper could not be expressive enough to model all kinds of dependencies found in web APIs. However, several reasons make us confident of the expresiveness of the language. First, IDL is partially inspired by the grammar of PICT, a mature combinatorial testing tool developed by Microsoft. Second, IDL is based on the findings of a thorough study of over 600 dependencies found in more than 2.5K operations. Finally, and more importantly, we were able to model a total of 171 new dependencies from 23 real-world APIs, without identifying expresiveness issues.

Finally, our work lacks an empirical validation with software developers and practitioners that ensures the usefulness and usability of the developed tools. IDL might be considered hard to understand or to familiarise with. To minimise this threat, the language provides syntactic sugar to make dependencies self-explanatory (i.e., *Or*, *OnlyOne*, *AllOrNone* and *ZeroOrOne*). Also, we have proposed IDL4OAS, which allows to succinctly specify inter-parameter dependencies in OAS, the de-facto standard for API specification in industry.

8 Related Work

To the best of our knowledge, this work is the first to fully address both the specification and the automated analysis of inter-parameter dependencies in web APIs. Therefore, we discuss the related work from these two different perspectives.

8.1 Specification

Oostvogels et al. [31] proposed OAS-IP, a DSL for the description of inter-parameter constraints in OAS. They classified dependencies into three types: *exclusive* (called *OnlyOne* in our work), *dependent* (*Requires* in our work), and *group constraints* (*AllOrNone* in our paper). For the design of the language, they studied six commercial APIs. Later on, they proposed TypeScript$_{\text{IPC}}$ [32], a TypeScript extension for static typing of constraints in interfaces in this specific programming language. Their work shares clear similarities with ours in terms of expressiveness, however, IDL stands out in three ways. First, IDL is specification-independent, and not tied to any programming language. Second, IDL is based on a more thorough analysis of industrial APIs and allows the description of dependency patterns not supported in OAS-IP: *Relational* and *ZeroOrOne*. Lastly, IDL provides syntactic sugar for all the dependency patterns found in practice, making it more succinct and less error-prone than OAS-IP. For example, a *ZeroOrOne* dependency in IDL (line 1 below) must be expressed with three composed dependencies in OAS-IP (lines 2-4):

```
1 ZeroOrOne(p1, p2, p3); // Equivalent in OAS-IP:
2 implic(present(p1), not(or(present(p2), present(p3))))
3 implic(present(p2), not(or(present(p1), present(p3))))
4 implic(present(p3), not(or(present(p1), present(p2))))
```

RAML, the *RESTful API Modeling Language* [38], provides basic support for inter-parameter dependencies. Mutually exclusive parameters (i.e., referred to as *OnlyOne* in our catalogue) can be specified thanks to the so-called *union* type, where a piece of data can be described by any of several types. For example, to describe two mutually exclusive parameters `p1` and `p2`, it could be done as follows: "`type: [p1 | p2]`". However, RAML does not support the remaining six dependency types presented in this article, which represent 83% of the dependencies found in our study of real-world web APIs [25].

A few proposals exist for the specification of dependencies in other types of web services such as WSDL [48]. Yang [52] and Xu et al. [51] leveraged the OWL-S [34] specification of the service to extract preconditions from them, such as relational dependencies among parameters. Cacciagrano et al. [5] used the Constraint Language in XML (CLiX) [8] to specify constraints in the WSDL specification of the service. Sun et al. [42] proposed WSDL with Constraints (CxWSDL), a language for specifying six common types of constraints of a service's implementation (e.g., a *time constraint*, for expressing the service availability). Other pieces of work [2], [4], [6] focus on the dependencies present among different operations, e.g., some operation must be invoked before some other for the service to work correctly. Compared to these approaches, our work focuses on inter-parameter dependencies, while they focus on behavioural or semantic constraints. At most, some contributions support the relational dependencies described in our catalogue [25]. Finally, these approaches are based on (and tied to) XML, making it difficult to integrate them in modern API specification languages like OAS.

Regarding the specification of dependencies, combinatorial test case generation tools offer similar capabilities to specify constraints among input parameters, e.g., TestCover [46], Advanced Combinatorial Testing System (ACTS) [56] and Pairwise Independent Combinatorial Testing (PICT) [35]. Unfortunately, these tools were not designed with reusability in mind and their use out of the context of testing is difficult. The syntax of IDL is partially based on that of

PICT, a fully-fledged tool developed by Microsoft. However, we extended the constraints grammar of PICT to support the seven types of dependencies from our catalogue [25], making IDL specifications succinct and self-explanatory.

8.2 Automated Analysis

Wu et al. [49] presented an approach for the automated inference of dependency constraints among input parameters in web services. They first established six dependency patterns, four of which are specific instances of the *Requires* dependency presented in our work, and then leveraged the service documentation, the SDK and the service itself to extract and validate candidates of dependencies. Compared to their approach, our work focuses on the specification and analysis of inter-parameter dependencies, and not in their inference. In this sense, their contributions are tangential to ours, and it should be possible to combine them. For example, we could use their approach to try to infer dependencies in APIs, based on our catalogue of dependency patterns.

Regarding WSDL and other XML-based specification formats, several authors have addressed the automated analysis of dependencies to some extent. The most related papers [5], [51], [52] leverage the service specification to generate requests satisfying or violating the constraints specified in it (similarly to our *random request* operation). Gao et al. [14] integrated information about parameters, error messages and testing results to infer data preconditions on web APIs that sometimes are not correctly specified in their documentation. Compared to these papers, we present a catalogue of seven novel analysis operations, and we propose a CSP-based implementation.

Regarding the automated analysis of IDL specifications, our proposal is inspired by previous work by the authors in the context of feature models, where more than 30 different analysis operations have been proposed [3]. Also, we were pioneers on the automated analysis of service level agreements in different web service technologies such as WS-Agreement [27], [28], [29], Linked USDL [15] and recently in OAS [13]. Both lines of research served as the basis for the contributions presented in this paper where, although with similar ideas, we had to face unique challenges due to the richer catalogue of dependency patterns found in web APIs.

9 Conclusions

This article addressed the problem of specifying the dependencies among input parameters in web APIs. We presented a domain specific language, IDL, specifically designed to express the seven types of dependencies observed in a thorough study of industrial APIs. Besides this, we proposed a catalogue of seven analysis operations to extract helpful information from IDL specifications such as detecting inconsistencies or checking the validity of API requests. For the automation of the analysis operations we proposed translating IDL specifications to CSPs and leveraging the capabilities of state-of-the-art CSP solvers. The approach is supported by an (Eclipse) editor, a parser, an OAS extension, an analysis library (IDLReasoner), and a test suite. Among its many applications, we showed the potential of IDLReasoner in the context of automated testing of RESTful APIs, revealing failures in three commercial APIs. Together, these contributions not only provide a complete solution to the automated management of inter-parameter dependencies in web APIs, but they also open a new range of applications and research opportunities in areas such as code generation, monitoring and testing.

Acknowledgements

This work has been partially supported by the European Commission (FEDER) and Junta de Andalucia under projects APOLO (US-1264651) and EKIPMENT-PLUS (P18-FR-2895), and by the Spanish Government under project HORATIO (RTI2018-101204-B-C21) and the FPU scholarship program, granted by the Spanish Ministry of Education and Vocational Training (FPU17/04077). We would also like to thank Roberto Hermoso for his technical support during the development of IDLReasoner.

References

[1] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *Intern. Conference on Software Engineering*, 2019, pp. 748–758.

[2] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-Based Automatic Test Case Generation for Web Services Testing," in *IEEE International Workshop on Service-Oriented System Engineering*, 2005, pp. 207–212.

[3] Benavides, D., Segura, S., Ruiz-Cortés, A., "Automated Analysis of Feature Models 20 Years Later: A Literature Review," *Information Systems*, vol. 35, no. 6, pp. 615 – 636, 2010.

[4] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic Synthesis of Behavior Protocols for Composable Web-Services," in *Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2009, pp. 141–150.

[5] D. Cacciagrano, F. Corradini, R. Culmone, and L. Vito, "Dynamic Constraint-based Invocation of Web Services," in *3rd Intern. Workshop on Web Services and Formal Methods*, 2006, pp. 138–147.

[6] A. Chaturvedi and D. Binkley, "Web Service Slicing: Intra and Inter-Operational Analysis to Test Changes," *IEEE Transactions on Services Computing*, 2018.

[7] "The Chuffed CP solver," accessed January 2020. [Online]. Available: https://github.com/chuffed/chuffed

[8] "CLiX - A Validation Rule Language for XML," accessed July 2020. [Online]. Available: https://www.w3.org/2004/12/rules-ws/paper/24/

[9] "Inter-Parameter Dependencies in RESTful APIs [Dataset]," 2019. [Online]. Available: https://bit.ly/2wvv1m1

[10] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[11] "Forte API," accessed January 2020. [Online]. Available: https://restdocs.forte.net/

[12] "Foursquare API," accessed January 2020. [Online]. Available: https://developer.foursquare.com/places-api

[13] A. Gamez-Diaz, P. Fernandez, and A. Ruiz-Cortés, "Governify for APIs: SLA-Driven Ecosystem for API Governance," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, p. 1120–1123.

[14] C. Gao, J. Wei, H. Zhong, and T. Huang, "Inferring Data Contract for Web-based API," in *IEEE Intern. Conference on Web Services*, 2014, pp. 65–72.

[15] J. García, P. Fernandez, C. Pedrinaci, M. Resinas, J. Cardoso, and A. Ruiz-Cortés, "Modeling Service Level Agreements with Linked USDL Agreement," *IEEE Transactions on Services Computing*, vol. 10, no. 1, pp. 52–65, 2017.

[16] "GECODE - An open, free, efficient constraint solving toolkit," accessed January 2020. [Online]. Available: https://www.gecode.org/

[17] "GitHub API," accessed January 2020. [Online]. Available: https://developer.github.com/v3/

[18] "Google Maps API," accessed January 2020. [Online]. Available: https://developers.google.com/places/web-service/intro

[19] "Inter-parameter Dependency Language (IDL)," accessed January 2020. [Online]. Available: https://github.com/isa-group/IDL

[20] "IDLReasoner: An Analysis Library for IDL Specifications," accessed January 2020. [Online]. Available: https://github.com/isa-group/IDLReasoner

[21] D. Jacobson, G. Brail, and D. Woods, *APIs: A Strategy Guide*. O'Reilly Media, Inc., 2011.

[22] D. Jacobson and S. Narayanan, "Netflix API: Top 10 Lessons Learned," in *Open Source Convention (OSCON)*, Porland, Oregon, July 2014. [Online]. Available: http://www.slideshare.net/danieljacobson/top-10-lessons-learned-from-the-netflix-api-oscon-2014

[23] "Last.fm API," accessed January 2020. [Online]. Available: https://www.last.fm/api

[24] "Language Server Protocol," accessed January 2020. [Online]. Available: https://microsoft.github.io/language-server-protocol

[25] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs," in *Intern. Conference on Service-Oriented Computing*, 2019, pp. 399–414.

[26] "MiniZinc: Constraint Modeling Language," accessed November 2019. [Online]. Available: https://www.minizinc.org/

[27] C. Müller, A. M. Gutierrez Fernandez, P. Fernandez, O. Martin-Diaz, M. Resinas, and A. Ruiz-Cortes, "Automated Validation of Compensable SLAs," *IEEE Transactions on Services Computing*, 2018, article in press.

[28] C. Müller, M. Resinas, and A. Ruiz-Cortés, "Automated Analysis of Conflicts in WS–Agreement," *IEEE Transactions on Services Computing*, vol. 7, no. 4, pp. 530–544, 2014.

[29] C. Müller, M. Oriol, X. Franch, J. Marco, M. Resinas, A. Ruiz-Cortés, and M. Rodríguez, "Comprehensive Explanation of SLA Violations at Runtime," *IEEE Transactions on Services Computing*, vol. 7, no. 2, pp. 168–183, 2014.

[30] "GitHub issue - Support interdependencies between query parameters," accessed July 2020. [Online]. Available: https://github.com/OAI/OpenAPI-Specification/issues/256

[31] N. Oostvogels, J. De Koster, and W. De Meuter, "Inter-parameter Constraints in Contemporary Web APIs," in *17th Intern. Conference on Web Engineering*, 2017, pp. 323–335.

[32] ——, "Static Typing of Complex Presence Constraints in Interfaces," in *32nd European Conference on Object-Oriented Programming*, 2018, pp. 1–27.

[33] "OpenAPI Specification," accessed March 2019. [Online]. Available: https://github.com/OAI/OpenAPI-Specification

[34] "Semantic Markup for Web Services (OWL-S)," accessed November 2019. [Online]. Available: https://www.w3.org/Submission/OWL-S/

[35] "Microsoft PICT - Pairwise Independent Combinatorial Testing," accessed October 2019. [Online]. Available: https://github.com/microsoft/pict

[36] "ProgrammableWeb API Directory," accessed March 2019. [Online]. Available: http://www.programmableweb.com/

[37] "QuickBooks Payments API," accessed January 2020. [Online]. Available: https://developer.intuit.com/app/developer/qbpayments/docs/get-started/

[38] "RESTful API Modeling Language (RAML)," accessed March 2019. [Online]. Available: http://raml.org/

[39] "RapidAPI API Directory," accessed March 2019. [Online]. Available: https://rapidapi.com

[40] "RESTest: Automated Black-Box Testing of RESTful Web APIs," accessed July 2020. [Online]. Available: https://github.com/isa-group/RESTest

[41] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[42] C.-a. Sun, M. Li, J. Jia, and J. Han, "Constraint-Based Model-Driven Testing of Web Services for Behavior Conformance," in *Intern. Conference on Service-Oriented Computing*, 2018, pp. 543–559.

[43] "Supplementary material of the paper." [Online]. Available: https://isa-group.github.io/2020-02-inter-parameter-dependencies

[44] "Swagger," accessed March 2019. [Online]. Available: http://swagger.io/

[45] A. H. M. Ter Hofstede and H. A. Proper, "How to Formalize It? Formalization Principles for Information System Development Methods," *Information and Software Technology*, vol. 40, pp. 519–540, 1998.

[46] "Testcover.com," accessed January 2020. [Online]. Available: https://www.testcover.com/

[47] "Twilio API," accessed January 2020. [Online]. Available: https://www.twilio.com/docs/usage/api/

[48] "Web Services Description Language (WSDL) Version 2.0," accessed November 2019. [Online]. Available: https://www.w3.org/TR/wsdl20/

[49] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Inferring Dependency Constraints on Parameters for Web Services," in *Proceedings of the 22nd Intern. Conference on World Wide Web*, 2013, pp. 1421–1432.

[50] "Xtext," accessed October 2019. [Online]. Available: https://www.eclipse.org/Xtext/index.html

[51] L. Xu, Q. Yuan, J. Wu, and C. Liu, "Ontology-based Web Service Robustness Test Generation," in *IEEE Intern. Symp. on Web Systems Evolution*, 2009, pp. 59–68.

[52] B. Yang, "Dependence Analysis for Web Services Data Mutation Testing," in *Advanced Multimedia and Ubiquitous Engineering*, 2016, pp. 761–768.

[53] "Yelp API," accessed January 2020. [Online]. Available: https://www.yelp.com/developers/documentation/v3

[54] "Twitter API," accessed January 2020. [Online]. Available: https://developer.twitter.com/en/docs

[55] "YouTube Data API v3," accessed January 2019. [Online]. Available: https://developers.google.com/youtube/v3/

[56] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "ACTS: A Combinatorial Test Generation Tool," in *International Conference on Software Testing, Verification and Validation*, 2013, pp. 370–375.

Alberto Martin-Lopez is a PhD candidate at the Applied Software Engineering research group (ISA, www.isa.us.es), University of Seville, Spain. He received his MsC from this university. His current research interests focus on automated software testing and service-oriented architectures. He is a Fulbright fellow and the winner of the ACM Student Research Competition held at ICSE 2020.

Sergio Segura is an Associate Professor of software engineering at the University of Seville, Spain. He is a member of the Applied Software Engineering research group, where he leads the research lines on software testing and search-based software engineering. His current research interests include test automation and AI-driven software engineering. Contact him at sergiosegura@us.es.

Carlos Müller is a Lecturer and member of the Applied Software Engineering Group (ISA, www.isa.us.es) at University of Sevilla, Spain. He obtained his PhD in Computer Science from this university. His current research line includes the automated analysis of service level agreements (SLA) and the application of such analysis at SLA design and monitoring.

Antonio Ruiz-Cortés is a Full Professor of software and service engineering and elected member of the Academy of Europe. He heads the Applied Software Engineering Group at the University of Sevilla. His current research focuses on service-oriented computing, business process management, testing and software product lines. He is an associate editor of Springer Computing. Contact him at aruiz@us.es.