API Description-Based Conformance Assessment of Architectural Design Decision

1st Apitchaka Singjai
Research Group Software Architecture
University of Vienna
Vienna, Austria
apitchaka.singjai@univie.ac.at

2nd Uwe Zdun
Research Group Software Architecture
University of Vienna
Vienna, Austria
uwe.zdun@univie.ac.at

***Abstract*—Microservice APIs are often designed based on Domain-Driven Design. It can be challenging to judge the quality of the relation between such a design and the implemented API, especially when facing frequent releases and changes in an API and its design models. Manual conformance assessment of this relation is time-intensive and error prone. This paper proposes a novel approach for automated conformance assessment of API designs in relation to API design decisions. Our approach aims to provide the first fully automated conformance assessment approach based solely on the code of API Descriptions such as OpenAPI. We applied and exemplified our approach for checking conformance to the patterns and practices in an API design decision for mapping links in a system's domain model to API representations. The total accuracy score (F1-measure) for decision option identification in our multi-case study for this decision is 97.22%.**

***Index Terms*—API Design, ADD, Architecture Conformance Assessment, DDD, Microservice Architecture.**

I. Introduction

Microservice architectures consist of independently deployable, scalable, and changeable services [1]–[3]. API design is an important aspect of microservice architectures and includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to offer links, and how to represent API messages [2], [4]. Many microservices and API abstractions are identified using Domain-Driven Design (DDD) [5]. DDD is a design approach that places the (business) domain at the center of software designing and architecting. In this paper, we focus on a specific aspect of mapping domain models to APIs: We apply and exemplify our approach using an Architectural Design Decision (ADD) on link mapping.

Supporting rapid release cycles is one of the key goals of many microservices architectures. Development teams want to deliver their APIs through a Continuous Integration/Continuous Delivery (CI/CD) pipeline. Continuous updates of APIs can quickly lead to violations of conformance of ADDs being taken in API design. Manual checking for such violations is error-prone and time-consuming. In general, such conformance evaluation has been used in a variety of domains of software engineering, including service composition [6] and conformance to standards [7]. The conformance relation is often described as the consistency of models [6]. In our context, architectural conformance assessment is used to verify that ADDs are implemented appropriately in a particular API endpoint. API Descriptions, such as OpenAPI[1], Swagger[2], or RAML[3], are widely used today. Our approach aims to provide the first fully automated conformance assessment based solely on the code of API Descriptions (we focus on OpenAPI). We set out to answer the following research questions:

- **RQ1** How to provide a fully automated conformance assessment of API ADDs?
- **RQ2 a)** To which extent can OpenAPI-based parsing provide sufficient information for design option identification in API ADDs? **b)** Which level of accuracy can be expected in such fully automated conformance assessment of API ADDs?

To address the research questions, we obtained empirically validated scoring schemes for the exemplary ADD LMD from our previous research. We then derived assessment algorithms and realized novel OpenAPI parser and assessor tools for conformance assessment. In parallel, we selected and analyzed case studies. Then we conducted a manual identification for ground truth assessment in our multi-case study. After that, we executed the automatic parsing and analysis of the case studies. Finally, we performed an empirical validation. For this, firstly, we computed accuracy measures (Precision, Recall, and F1-measure) for the identification of decision options compared to our ground truth. Secondly, we calculated the simple count comparison metric between the correct identification of conformance results and the number of API endpoints.

To address RQ1, we built the OpenAPI parser and assessor tools to automate the following tasks: parsing the OpenAPI description, identifying decision options, and assessing conformance. Regarding RQ2, we reached an overall average accuracy score of 97.22 (F1-measure) in our multi-case study.

These numbers seem to indicate that the OpenAPI descriptions (in the cases and for the selected ADD) provide enough information for an acceptable automatic conformance identification.

[1]https://www.openapis.org/
[2]https://swagger.io/
[3]https://raml.org/

This paper is organized as follows. Section II we describe our motivation and background. Section III we discuss our research methods. Then, we explain how we prepared the case studies in Section IV. In Section V, we present the automated approach for assessing the conformance, and present the empirical validation results in Section VI. After that, we discuss our results in Section VII. We compare to related works in Section VIII. Finally, we discuss threats to validity in Section IX and draw conclusions in Section X.

II. BACKGROUND

The primary motivation for this work can be linked to the results from our earlier empirical study of practitioners' perspectives on the interrelations between APIs and DDD [8]. In this work, we empirically identified multiple ADDs in API design that contains API design best practices as decision options. We selected one ADD from this work, the Link Mapping Decision (LMD), to exemplify and validate our work in this paper. The Link Mapping Decision (LMD) was selected because it proved to be the hardest to assess automatically [9].

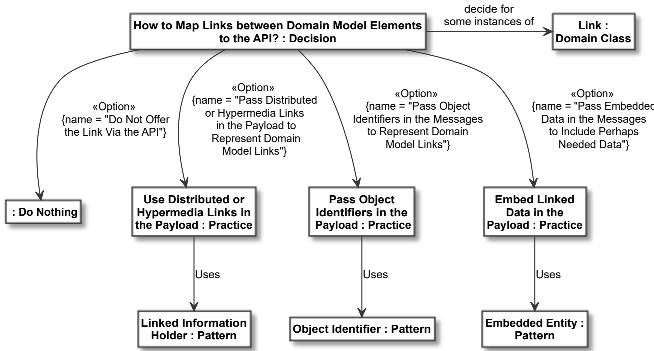

Fig. 1: Link Mapping Decision

The Link Mapping Decision (LMD) describes possible options for the connections between API endpoints. Links between model elements in a domain model describing the API have a similar function in DDD. Obviously, not all links in a DDD model are eligible for API exposure, but only those between model elements that are designated to be exposed in API endpoints. The Link Mapping Decision is shown in Figure 1 along with its alternative decision options. A link in a domain model to be exposed to an API endpoint can be mapped in the following possible ways: Firstly, the option *Use Distributed or Hypermedia Links in the Payload* maps the domain model link to a standard distributed or hypermedia link, such as URIs in RESTful HTTP or URIs and HAL/JSON-LDs in JSON. An alternative is *Embed Linked Data in the Payload* where the content is added to the message payload rather than connecting to it via a distributed link. Next, there is the option to *Pass Object Identifiers in the Payload* which sends an *Object Identifier* that is meaningful (only) in the server context, but contains no remote location information. Note that each of these options is linked to an API Design Pattern described in the literature, namely LINKED INFORMATION HOLDER [4], OBJECT IDENTIFIER [4], [10], and EMBEDDED ENTITY [4]. Finally, there is the option to choose not to map the domain model link to the API.

The choice of an option has positive or negative influences on desired API qualities. For example, compared to the EMBEDDED ENTITY option, use of distributed links is beneficial for *Data Consistency* as the link is always up-to-date, and thus *API Evolvability* and *API Modifiability* are positively influenced. It leads also to smaller *Message Sizes*. Links however lead to higher *Protocol Complexity*, making it harder to *Minimize API Calls*, and can have a worse *Performance* and *Scalability* because of many resulting distributed calls. The *Object Identifier* based option is very similar in its effects to the distributed links based option. In direct comparison, it has the disadvantages of possibly *Exposing Domain Model Details in API* as well as higher *Coupling of Clients to Server*.

In our previous work [9], we derived an *ordinal scoring scheme* for manual assessment of an API—Domain Model mapping. It is our goal in this work, to automate the human judgement in this scoring scheme solely based on the OpenAPI description of an API. The scoring scheme reflects the empirical insights how practitioners judge different combinations of the above explained decision options:

- IF FOR SOME domain links which are needed by the clients: *Do not offer the Link via the API* THEN assessment = (- -).
- IF (FOR ALL domain links dl which are needed by the clients: (IF for dl clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* is used)) THEN assessment = (++).
- IF ALL domain links dl which are needed by the clients: (IF For dl clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* OR *Pass Object Identifiers in the Payload* is used)) THEN assessment = (+).
- IF (FOR SOME domain links dl which are needed by the clients: If for dl clients usually require most of the linked data elements immediately: *Embed Linked Data in the Payload* is NOT used for dl) THEN assessment = (-)
- IF any other combination is used (e.g. for some domain links dl which are needed by the clients, and for dl clients usually do NOT require most of the linked data elements immediately AND *Embed Linked Data in the Payload* is used for dl) THEN assessment = (o).

In general, our conformance assessment approach is based on the notion of a structured API description or API contract [11] as input. API DESCRIPTION [12] is a pattern that was previously released as part of the Microservice API Patterns [4], and API CONTRACT is a variant of the general INTERFACE DESCRIPTION pattern [10].

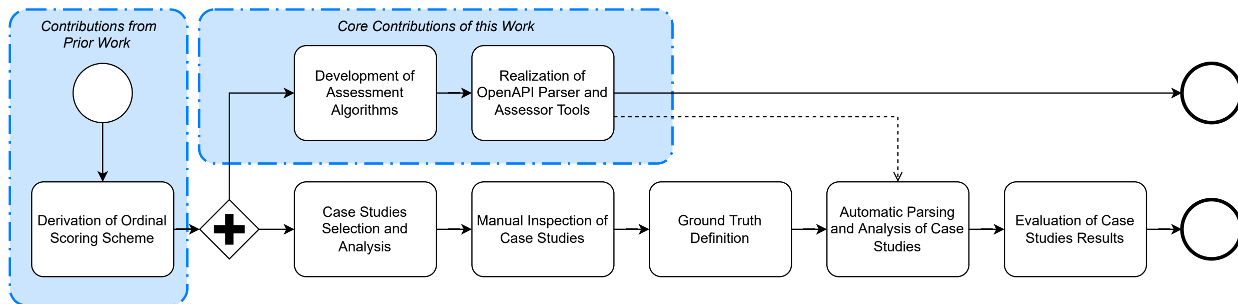

Fig. 2: Research Methodology Overview

III. Research Methodology

Figure 2 illustrates the research methods applied in our study. The *derivation of ordinal scoring scheme* from the previous section has been performed in previous work [9]. Github[4] served as the primary source for *case studies selection and analysis*. We followed the recommendations established by Petersen et al. [13] (originally for systematic mapping studies) for establishing search phrases according to the PICO principles [14]. We applied two principles from PICO, which are *population* and *intervention*. Then, we applied inclusion/exclusion rules. The case studies were chosen independently and in parallel to our approach's development. We performed a *manual inspection of case studies* and created the *ground truth definition* for each API endpoint in the case studies. This ground truth is based on the scoring scheme from the previous work. We have followed the following definitions during our inspections [4], [12], [15]: An *API endpoint* is the provider-side end of a communication channel and a specification of where the *API* endpoints are located so that *APIs* can be accessed by *API clients*. Each endpoint thus needs to have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), as well as in HTTP-based SOAP or RESTful HTTP. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints.

For *Automatic Parsing and Analysis of Case studies*, we applied the Panda and Numpy libraries. We compared the results of the automatic assessment with the ground truth assessments in our *Evaluation*. We quantified our findings by counting the correct assessment identifications and calculating accuracy scores.

In the following subsections, we explain the methods for each of these core contributions.

A. Realization of OpenAPI Parser and Assessor Tools

We constructed the parsing and assessment algorithms using our prior work's scoring scheme. While the conditions remained constant, the scope of identification shifted. Rather than analyzing the whole system, we opted to assess individual API endpoints. Then, we created an algorithm that receives the endpoint name, HTTP methods, and the binary value of each decision option. It returns the assessment result and the endpoint name. We used the OpenAPI specification as a starting point to identify three kinds of relations:

1) single decision option per data elements,
2) multiple decision options per operation, and
3) the existence of decision options per API endpoint.

We analyze these three relations in turn. To identify the operation, we used the *operationId* element of OpenAPI-based specification in the first place. When the *operationId* is omitted; we inspect the HTTP method instead. To obtain this relation, we use a parser-based detection technique. We adapted the swagger-reader [16] for this purpose. Our OpenAPI parser converts the information into a decision option identification. Moreover, our output identifies the interrelation between the set of decision options and the API operation.

a) Decision Options Identification and Verification: For the decision option identification, we started by applying the parser-based detection approach. The result is the automated generation of the option per API data element (Relation 1 in the previous section). For validation, we generated the raw accuracy data (True Positives: TP, True Negatives: TN, False Positives: FP, and False Negatives: FN) by comparing the automated detection with the manual detection. Then, we calculated the accuracy scores (Precision, Recall, and F1-Measure) using the following equations:

$$Precision = \frac{\sum_{n=1}^{\infty} TP}{\sum_{n=1}^{\infty}(TP + FP)} \quad (1)$$

$$Recall = \frac{\sum_{n=1}^{\infty} TP}{\sum_{n=1}^{\infty}(TP + FN)} \quad (2)$$

$$F1 - Measure = 2 * \left[\frac{Precision * Recall}{Precision + Recall}\right] \quad (3)$$

IV. Case Studies Selection and Analysis

This section elaborates on how we selected the case studies and what the ground truth assessments of each case study are that resulted from our manual assessment of the cases.

Firstly, we determined the search string keywords. Our population was restricted to the Github repositories. The interventions are the API DESCRIPTION pattern and OpenAPI. As a result, we utilized the Github Search API to do the queries based on a number of criteria:

[4] https://github.com/

- The search string (“OpenAPI”, “API”, “specification”) in project name, description, and/or README file.
- The repository’s size in the rank 10-100 MB.
- The author has more than 9 followers.
- The search string(“OpenAPI”, “API specification”) without any conditions.

As a result, we obtained 83 repositories in total from the Github repository. Then, we subjected all sources to an in-depth investigation. The following inclusion/exclusion criteria were used to include sources:

- Sources with OpenAPI specifications: We reviewed each source individually and rejected sources that did not use OpenAPI.
- Sources with API specifications relevant to the project itself: we examined each API specification and filtered out test files, package files, sample files, example files, configuration files, and generated API specification files.
- Sources with valid API specifications: We used the OpenAPI Tree tool[5] to test whether or not an API specification was legitimate. A graphical representation, such as the one provided in Section V, is generated when the API specification is valid. We also included sources where the API specification came with API contracts and the ability to assess the LMD was given, even if the OpenAPI Tree tool failed.
- Sources with more than 500 SLOCs.

Moreover, we included two sources from the prior case studies that met these criteria. They came with API contracts and the ability to assess the LMD as well. The Table I shows a brief summary of each of the inspected cases studies, ID, number of slocs, number of endpoints, number of pairings between API endpoint/API operation, and the number of pairings between API endpoint/API data element.

To provide a baseline for further assessment, we manually analyzed each API endpoint in our multi-case studies based on the scoring scheme from Section II. Regarding the ground truth agreement, one author used the scoring scheme to derive the ordinal scale of each API endpoint. Then, the other author verified these judgements. Table II summarizes the ground truth assessments. Please note that this is not the result of the automated assessment approach presented below but the manually derived ground truth obtained by manually assessing every single endpoint of the 7 cases. We use it later for scientific evaluation of the automatically computed results. As can be seen, the cases cover a wide range of possible combinations of the decision options for the decisions.

V. Proposed Approach

This section elaborates on the proposed approach, i.e. how API developers would use our approach. Our approach starts by parsing an API description and then performs an identification of the decision options. In particular, we first identify the decision option chosen in the API description per API data element, then per API operation, and then per API endpoint. Three levels of step-wise decision option identification are needed to reflect the complexity of the decision’s relation, as explained in Section II.

[5]http://api-ace.inf.usi.ch/openapi-to-tree/

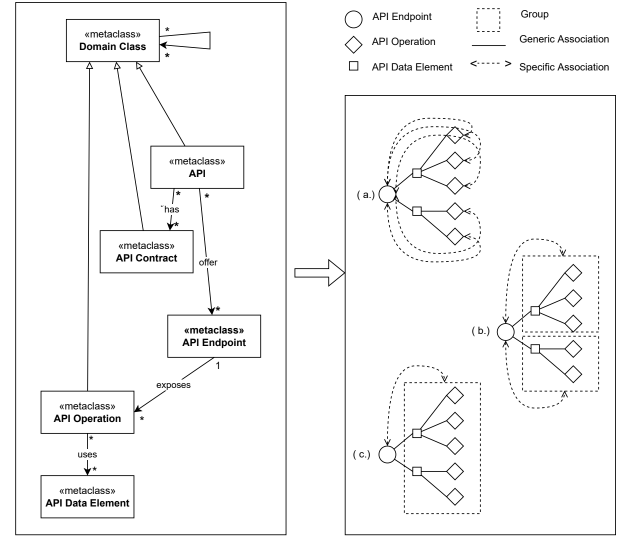

Fig. 3: Three Levels of Step-wise Identification

Figure 3 shows an overview of the three levels of step-wise decision option identification. To illustrate these levels, we selected three API meta-model elements—*API Endpoint*, *API Operation*, and *API Data Element*—that are shown on the left-hand side of the figure. We show these meta-model elements in Figure 3 in relations using three kinds of associations: *Group*, *Generic Association*, and *Specific Association*. One API endpoint, two API operations, and five API data elements are depicted in Figure 3 (a.)-(c.). They depict the relationships between API endpoints/API data elements, API endpoints/API operations, and API endpoints with six, two, and one pairs between the API endpoint and its data elements, respectively. Our three identification levels are used to identify each of these possible structures in OpenAPI.

Based on the detected information, the approach then assesses the conformance using the scoring scheme. Finally, it generates the assessment results to be displayed to the API developer. Please note that these steps are to be conducted entirely based on OpenAPI-based API descriptions, i.e. requiring no human effort for modeling creation.

Figure 4 shows the tree model of the *Real World Example App* case’s OpenAPI specification, which we use as a running example. It contains 12 API endpoints for which we evaluated LMD conformance. The decision option chosen for each endpoint depends on the relationship between the API endpoint and the API data elements used in its operations. Since we studied both the incoming and outgoing data elements in the API specification, we have 61 pairings between the API endpoint and its data elements. Further, there are 19 relations between the API endpoints and their operations. To illustrate this further, we use the endpoint “/profiles/{username}/follow” highlighted in red in Figure 4 as a running example.

TABLE I: Overview of the Inspected Case Studies

ID	Description	(#) sloc	(#) Endpoint	(#)Endpoint -operation	(#)Endpoint -data
RW	Realworld Example App: The API specification supports technology stack diversity. The documentation of a full-stack app called Conduit is available in the YAML format (version 3.0.1). Ref:https://github.com/gothinkster/realworld	916	12	19	61
DX	Graph DevX API: It provides a backend RESTful API that has resources used by the Microsoft Graph documentation, Graph Explorer, Powershell SDK, and Graph samples workload teams, all of which are handled by the Graph PM team. Ref: https://github.com/microsoftgraph/microsoft-graph-devx-api	875	10	11	55
NC	Disease Statistics App: It delivers a variety of virus-related information. Its public API is an open-source project that is spearheaded by four primary contributors from four different countries. The specification for is available in JSON format (version 3.0.0). Ref: https://github.com/disease-sh/API)	2773	39	39	107
EA	Admin Example API Description: It provides the activity of the admin. The current version of the OpenAPI specification is 3.0.1. Ref: https://github.com/mohsenTalal/OpenAPI-Swagger	865	7	10	95
PT	Pinterest's REST API OpenAPI Description: This description is utilized internally by Pinterest's API architecture. It contains OpenAPI specifications in version 3.0.3 Ref: https://github.com/pinterest/api-description	4981	23	30	190
RG	Registry API: It enables teams to maintain API descriptions. The Registry API is a database of business APIs that is machine-readable and serves as the basis for web directories, portals, and workflow managers. Ref:https://github.com/apigee/registry	1846	20	35	203
ST	STITCH API Description: STITCH is a database that projected chemical-protein interactions. Correlations are formed both directly (physically) and indirectly (functionally) through computer prediction, information transfer between species, and aggregated interactions from other (primary) databases. Ref: https://github.com/micheldumontier/API-descriptions	716	12	12	65

TABLE II: Ground Truth Assessment: Number of Results per Ordinal Score in each Case Study

Assessment Results	RW	DX	NC	EA	PT	RG	ST
+ +	5	6	28	0	5	0	0
+	0	0	0	0	3	6	0
o	5	3	0	7	13	11	12
-	2	1	11	0	1	3	0
- -	0	0	0	0	0	0	0

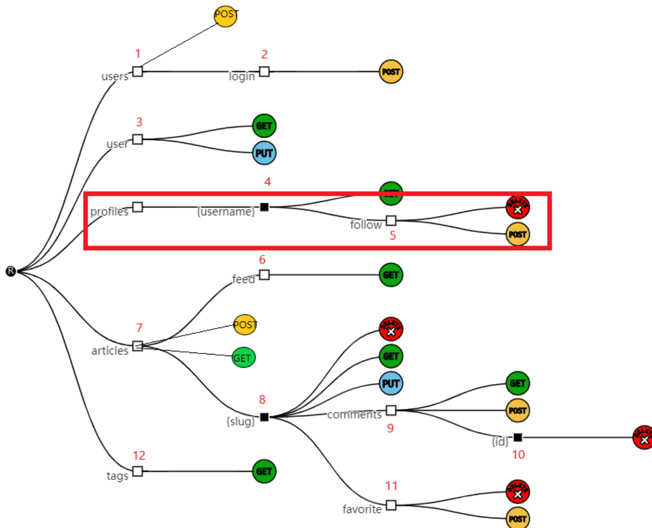

Fig. 4: OpenAPI Tree of the *Real World Example App* case

A. Parsing API Description

For the LMD decision, we parse the following information using our OpenAPI Parser: 1.) *Endpoint Data* 2.) *Operation ID* 3.) *Incoming Distributed Link* 4.) *Incoming Embedded Data* 5.) *Incoming Object ID Based Link* 6.) *Response Code* 7.) *Outgoing Distributed Link* 8.) *Outgoing Embedded Data* 9.) *Outgoing Object ID Based Link*.

The 1.) *Endpoint Data* here contains the API endpoint's name and the HTTP method. Since one API endpoint can bind with more than one HTTP method, we also included the HTTP method to make this endpoint data be usable as an identifier. The HTTP method identification, performed here, is helpful for API data element identification later on. The 2.) *Operation ID* helps to check whether the operation is already implemented in the backend, while the 6.) *Response Code* helps to check the availability of the outgoing data.

The three incoming data patterns, 3.) *Incoming Distributed Link*, 4.) *Incoming Embedded Data*, 5.) *Incoming Object ID Based Link*, as well as the three outgoing data patterns, 7.) *Outgoing Distributed Link*, 8.) *Outgoing Embedded Data*, 9.) *Outgoing Object ID Based Link*, are the specific information needed for identifying the LMD decision options (using the approach explained above).

Table III shows the sample output that we retrieve from the OpenAPI Parser for the *Real World Example App* study case. This table is also the sample input of the assessor in Section V-C.

B. Identification of Decision Options

There are 916 SLOCs in RW's OpenAPI specification. The code snippet of the endpoint "/profiles/{username}/follow" using the POST HTTP method is shown in Listing 1. Listing 1 contains one API endpoint (/profiles/{username}/follow), one relation between API endpoint and API operation (/profiles/{username}/follow - POST or FollowUserByUsername), and three relations between API endpoint and API data element that we used to identify the decision option as follows:

- /profiles/{username}/follow - username
- /profiles/{username}/follow - ProfileResponse

TABLE III: Sample Output for *Real World Example App* from the OpenAPI Parser

ID	1	2	3	4	5	6	7	8	9
RW-A1	/profiles/{username}/follow-DELETE	UnfollowUserByUsername	null	username:true	null	200—401—422	application/json: #/components/schemas/ ProfileResponse| application/json: #/components/schemas/ GenericErrorModel	null	null
RW-A2	/profiles/{username}/follow-POST	FollowUserByUsername	null	username:true	null	200—401—422	application/json: #/components/schemas/ ProfileResponse| application/json: #/components/schemas/ GenericErrorModel	null	null

- /profiles/{username}/follow - GenericErrorModel

Listing 1 contains snippets that correspond to the detected options RW-V5 and RW-V7 in Table IV.

Listing 1: The API Description Example


```
"/profiles/{username}/follow": {
"post": {
  "tags": ["Profile"],
  "summary": "Follow a user",
  "description": "Follow a user by username",
  "operationId": "FollowUserByUsername",
  "parameters": [{
    "name": "username",
    "in": "path",
    "description":
      "Username of the profile you want to follow",
    "required": true,
    "schema": {"type": "string"}
  }],
  "responses": {
    "200": {
      "description": "OK",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/ProfileResponse"
          }
        }
      }
    },
    "422": {
      "description": "Unexpected error",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/GenericErrorModel"
          }
        }
      }
    },
```


Table IV presents the decision option identifications. The Columns *identifier* and *decision-option* are generated automatically. Moreover, we included an ID column for the reporting purpose. The six entries (RW-V1 to RW-V6) are all addressing the same API endpoint (/profiles/username/follow). The two operations are *UnfollowUserByUsername* and *FollowUserByUsername*. Even though, we can retrieve these operations' name from the "operationTd" element, the HTTP methods are interchangeable because the relation between the HTTP method and "operationId" elements is 1:1. There are three data elements involved in the example with these two operations: *GenericErrorModel*, *ProfileResponse*, and *username*. In conclusion, the API endpoint contains two relations between an API endpoint and API operations, and six relations between the API endpoint and API data elements.

Due to the fact that we have three solution patterns and we investigated all of them in both incoming and outgoing data, we have six different decision options to detect as shown in Table V. The table summarizes the concepts applied to retrieve the decision option from API description.

The input is the *OpenAPI* description. To collect information from OpenAPI, we adopted the swagger model and swagger parser libraries. The output is a hashmap of the *identifier* and the *decision option* as shown in Table A IV. We aimed to identify the decision options of LMD between an API endpoint and a single data element of this API endpoint. We investigated the *parameters* and *requestBody* elements for request data (incoming data), while the *responses* element was investigated for response data (outgoing data). The *parameters* element holds the information related to the GET, DELETE, and HEAD HTTP methods. The *requestBody* element contains the information related to the POST, PUT, and PATCH HTTP methods. Moreover, we added *checkKeywords()* to allow us identifying id-related keywords, such as "id" or "identifier". For DISTRIBUTED LINKS, the investigated sub-element is *§ref*. For EMBEDDED DATA and OBJECT ID LINKS, the syntax is almost identical. We checked the *name* and *description* sub-elements in the *parameters* element and the *item.name* sub-element in the *responses* element. The association between API endpoint and the API data element turns to the OBJECT ID LINKS decision option when *checkKeywords()* is positive, but the EMBEDDED DATA decision option is the opposite.

C. Conformance Assessment

To provide our assessment algorithm, we further needed to resolve the difficulty of identifying decision options through the parser-based approach. We were able to associate the multiple decision options per operation using the parser-based detection technique. To assess the conformance assessment per API endpoint, we focused on the existence of decision options. Concerning the scoring scheme in Section II, we interpreted the information for two conditions: C1.) domain links are needed by the clients or not? C2.) clients usually require most of the linked data elements immediately or not? The answer for C1 should be "need" or "no need", whereas the answer of C2 should be "yes" or "no". Based on this, we can then fully assess the conformance to one of the decision options

TABLE IV: Decision Options Detection

ID	IDENTIFIER	DECISION-OPTION
RW-V1	/profiles/{username}/follow;DELETE;UnfollowUserByUsername;application/json:#/components/schemas/GenericErrorModel	RES-DistributedLink
RW-V2	/profiles/{username}/follow;DELETE;UnfollowUserByUsername;application/json:#/components/schemas/ProfileResponse	RES-DistributedLink
RW-V3	/profiles/{username}/follow;DELETE;UnfollowUserByUsername;username:true	REQ-EmbededData
RW-V4	/profiles/{username}/follow;POST;FollowUserByUsername;application/json:#/components/schemas/GenericErrorModel	RES-DistributedLink
RW-V5	/profiles/{username}/follow;POST;FollowUserByUsername;application/json:#/components/schemas/ProfileResponse	RES-DistributedLink
RW-V6	/profiles/{username}/follow;POST;FollowUserByUsername;username:true	REQ-EmbededData

TABLE V: The rule of Decision Options Identification

Decision Option	**Request**		**Response**
element	*parameters*	*requestBody*	*responses*
DISTRIBUTED -LINK	$ref	$ref	$ref + checkResponse()
EMBEDDED -DATA	name, description + !checkKeywords()	x	item.name + !checkKeywords() + checkResponse()
OBJECT-ID -LINK	name, description + checkKeywords()	x	item.name + checkKeywords() + checkResponse()

("Distributed Link", "Embeded Data", "Object ID Link", and "None Link"). Due to the scoring scheme's complexity, we included the HTTP method to represent the operation since certain API endpoints in the API specification lacked an *operationID*.

The handling of a single decision option per data elements was explained in Section V-B. The judgment of LMD was based on the relation between the single data element and the API endpoint. The existence of decision options per API endpoint was explained in the Algorithm 1.

Algorithm 1: Assessment Pseudocode

```
input:  String api_description
output: DataFrame assessment_results
begin
        assessment_results = parseAPIDescription(api_description)
        assessment_results.groupByAPIEndpoint()
        assessment_results.transformOptionsToBoolean()
        addAssessmentColumn(assessment_results)
        return assessment_results
end
```

Algorithm 1 presents the assessment pseudocode. The input is the String from API description. The output is a DataFrame with assessment result. *addAssessmentColumn()* handles the multiple decision options per operation that have been identified by grouping the decision options based on their operation. We modified the prior parser-based detection rule to enable it to extract the association between the multiple decision options and the operation. Since one API Endpoint possibly contains multiple operations, we applied *groupByAPIEndpoint()* to reduce the investigated entries. These investigated entries are the information per one API endpoint. After that, we turn the decision options which is the String value in to the Boolean value using *transformOptionsToBoolean()*. Thus the existence of each decision option is denoted as TRUE or FALSE. For *addAssessmentColumn()*, the input—*assessment_results*—refers to the endpoint name, the HTTP method, and a collection of Boolean values. The set of Boolean values represent the existence of that particular decision options of each API enpdoint. The results are returned as a tuple.

As previously stated, the /profiles/username/follow API endpoint has two associations between the API endpoint and the operation. Table III shows these two associations, whereas Table VI shows the final outcome, the conformance assessment result of a single API endpoint.

VI. Empirical Validation Results

To empirically validate our approach, we conducted two empirical validations: the *verification of decision options* and the *comparison to the ground truth*. While the verification of decision options are measured by the accuracy scores, the comparisons to the ground truth are using a simple count metric.

a) Verification of Decision Options: Table VII summarizes the results for the seven API descriptions that we selected from the third-party sources in our multi-case study. Accuracy score information includes the number of inspected relations (ED), and the prediction outcome in terms of True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN). The number of inspected relations for the cases varied from 55 to 203. These relations refer to the links between the API endpoint and the single data elements. There are a total of 776 inspected relations (ED).

Consequently, the average Precision, Recall, and F1-Measure values are 0.9479, 1, and 0.9722, respectively. To provide an audit trail of the research and enable repeatability of the study, we provide open access to our the data set and the source code[6].

b) Comparison to the Ground Truth on the Scoring Scheme: This comparison is for the conformance assessment correctness of the LMD decision based on the extracted data are shown in Table VIII.

The endpoints of *Real World Example App* (RW), *The Graph DevX API* (DX), *Disease Statistics App* (NC), *Admin Example API Description* (EA), *Pinterest's REST API OpenAPI Specification* (PT), *The Register API* (RG), *STITCH API Description* (ST) are 12, 10, 39, 7, 23, 20, and 12, respectively. AR stands for an actual assessment, which denotes an actual assessment produced automatically by our assessor tool. On the other hand, ER stands for an expected result, which conforms to the ground truth. Table VIII summarizes the evaluation results for the seven case studies using simple count metrics, with an overall match of 95.12 %. This score was

[6]We provide generated data, Python source code, and two excutable java files with dependencies as a replication package for download in the Zenodo long-term open access archive (10.5281/zenodo.6564304)

TABLE VI: The Output of *Real World Example App* from the Assessor Tool

Index	API-Endpoint(1–)	HTTP-Method (–1)	3	4	5	7	8	9	Actual-Assessment	Expected-Assessment
...										
7	/profiles/{username}/follow	POST; DELETE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	Neutral	Neutral
...										

TABLE VII: Information of Accuracy Scores

ID	ED	TP	FP	FN	TN	P	R	F1
RW	61	61	3	0	306	0.9531	1	0.976
DX	55	55	4	0	271	0.9322	1	0.9649
NC	107	107	20	0	515	0.8425	1	0.9145
EA	95	95	0	0	475	1	1	1
PT	190	172	18	0	950	0.9503	1	0.9503
RG	203	203	0	0	1015	1	1	1
ST	65	65	0	0	325	1	1	1
Total	776	**Average Accuracy Scores**				0.9479	1	0.9722

TABLE VIII: Information of Accuracy Scores

Assessment Results		(+ +)	(+)	(o)	(-)	(- -)	Correct Identification API Endpoints' number
RW	AR	4	0	6	2	0	
	ER	5	0	5	2	0	11/12
DX	AR	7	0	2	1	0	
	ER	6	0	3	1	0	9/10
NC	AR	28	0	0	11	0	
	ER	28	0	0	11	0	39/39
EA	AR	0	0	7	0	0	
	ER	0	0	7	0	0	7/7
PT	AR	3	1	15	4	0	
	ER	5	3	13	2	0	19/23
RG	AR	0	6	11	3	0	
	ER	0	6	11	3	0	20/20
ST	AR	0	0	12	0	0	
	ER	0	0	12	0	0	12/12
Total							117/123 = 95.12%

determined by dividing the number of correct identifications by the number of endpoints. *Disease Statistics App*, *Admin Example API Description*, *The Register API*, and *STITCH API Description* are four case studies with perfect matching.

VII. Discussion

Our automated conformance assessment is intended to substitute the time-consuming manual assessment. Concerning the API design from a DevOps or CI/CD perspective, automated quality control tasks are beneficial in terms of ensuring quality and avoiding error-prone and time-consuming manual inspections. We provided the novel concepts for the OpenAPI parser and the assessor tools to address RQ1. The OpenAPI parser is used to generate the input for the assessor tool that reports the ordinary scale conformance assessment per API endpoint. These tools aid in the following manual tasks: parsing OpenAPI description, identifying the decision options, and assessing conformance. Additionally, we demonstrated an approach using the LMD ADD and *Real World Example App* case study as examples.

To answer RQ2a, we conducted an empirical study to validate the decision option as discussed in Section VI-0a above. Overall, with 97.22% the accuracy score (F1 measure) is quite high (substantially better than 0.8). Occasionally, Embedded Data is unrecognized by the parser-based detector. Because in OpenAPI the source of the root of the OBJECT ID and EMBEDDED DATA is the same, we may presume that they are not different. Please note that this problem may be resolved by assigning an array string datatype rather than a single string datatype. While this would require changing the OpenAPI specifications that are inspected, with a simple developer guideline and an automated test of its application, we could potentially achieve 100% accuracy in our cases.

To answer RQ2b, we conducted an empirical study to compare the automated results to the application of the scoring scheme in the ground truth. We also illustrate these comparison results of all seven API specifications with the stack bars in Figure 5. The stack bars show the assessment information for both the expected results (ER) and the actual results (AR). Our case studies include a wide range of endpoints, ranging from seven in the *Admin Example API Description* case study to 39 in the *Disease Statistics App* case study.

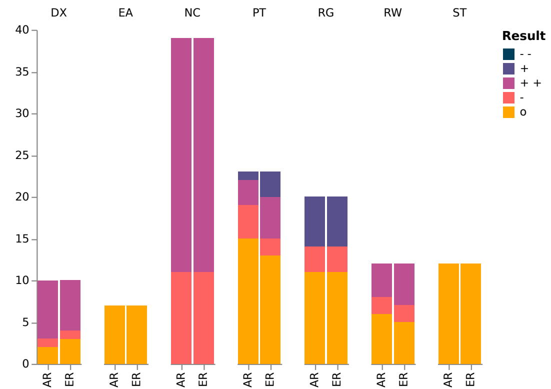

Fig. 5: The Comparison of Case Studies' Assessment Results

In the *Real World Example App* and *The Graph DevX API* case studies, one result does not match due to two API operations' involvement. We could achieve 100% matching in these scenarios by implementing further processing to analyze the relationship before combining the API operation element.

For the *Pinterest's REST API OpenAPI Specification* case study, four results are not matching. There are two reasons: 1.) from *Negative* to *Positive*: sometime, the OBJECT ID and EMBEDDED DATA can refer to the same item in certain cases. 2.) from *Neutral* to *Very Positive*: more caution is required when two operations are involved. Additionally, such mismatches could be caused in cases where just the ID-object is specified or where double linking is used to facilitate distributed links.

Four case studies got a fully correct match to the ground truth, namely *Disease Statistics App*, *Admin Example API Description*, *Registry API*, and *STITCH API Description*. Due to the fact that *Disease Statistics App* is composed entirely of GET HTTP methods, it is relatively simple to retrieve

a correct conformance assessment here. Also, this project's evaluation result has just two values: very positive (++) and negative (-). When clients require the majority of the linked data elements instantly, very positive means, "Embed Linked Data in the Payload" is utilized for the necessary linked data, unless "Distributed or Hypermedia Links in the Payload" is used. The negative value indicates that "Embed Linked Data in the Payload" is not utilized for certain domain connections required by clients. Both the *Admin Example API Description* and *STITCH API Description* case studies, even though their assessment results show only neutral values (o), allow clients to input the request in various forms ("application/json-patch+json", "application/json", "text/json", and "application/*+json"). Nevertheless, we treated it as one association between an API endpoint and a single API data element. For the RG, we added the "id." keyword to checkKeywords() since the authors of this API description often used "id." to indicate their OBJECT ID LINK. As a result, we were able to achieve a 100% of the matching score. This shows that sometimes minimal development effort is needed, like adding an identifier name, to achieve fully correct results.

In conclusion, 117 endpoints out of 123 were assessed correctly. The overall matching score is 95.12 percent.

VIII. Related Works

Three strategies exist for ensuring static architectural conformance: dependency-structure matrices, source code query languages, and reflexion models [17]. Architecture conformance or compliance checking has been employed to ensure architecture consistency [18]. Both terms are often used synonymously. Architectural documentation that preserves the architectural knowledge becomes irrelevant when the conformance between documentation and implementation is nonexistent [19]. This kind of conformance is constructed through model-driven engineering or reverse engineering. Some studies have been conducted on the conformance assessment of ADDs [20]–[22]. Zdun et al. [20] studied ensuring and assessing architecture conformance to microservice decomposition patterns. Their work consists of two main contributions: microservice design constraints and metrics to evaluate the architecture conformance to microservice patterns. Ntentos et al. [21] proposed a model-based technique for evaluating architectural conformance to microservice architecture patterns and practices. They also offered the metrics for assessing architecture conformance if an architecture complies with a typical architectural design option in the microservice domain [22]. In contrast to our approach, these approaches work from models; our approach only requires the OpenAPI code.

Some researchers studied conformance related to API or DDD. Athanasopoulos et al. [23] interpreted the framework for assessing interface uniformity in REST by discussing a conceptual framework and a criteria's collection. Domain-specific strategies might be used to map these criteria in terms of directing and/or inspecting the level of uniformity of a REST-based API. Kapferer and Zimmermann [24] proposed domain-driven architecture modeling and rapid prototyping with a tool called Context Mapper. They considered the conformance of the Domain-Specific Language (DSL) with the original DDD patterns. Our approach is the first to study conformance in the mapping between DDD models and APIs, ADDs in this context, and works by checking the API description code directly without human specification effort.

Concerning API design from a DevOps perspective, automated tasks are beneficial. Numerous tasks during the design phase are difficult to automate. The following related works, however, advocate for an automated API method over a manual one, similar to what we propose. Robillard et al. [25] surveyed automated API property inference techniques. They classify 60 various strategies into five groups based on their systematic review. Scheller and Kühn [26] proposed the API concepts framework for automated measurement of API usability. Sohan et al. are interested in automated API documentation. They introduced automatic RESTful API documentation using an HTTP proxy server [27].

IX. Threats to Validity

This section discusses threats to the validity of our study. Our ground truth assessment is dependent on how the ADDs are interpreted, and different practitioners might get slightly different conclusions. We mitigated this subjectivity by comparing the ground truth assessment in each iteration of our research study to the prior study [8]. In this empirical work, we considered a relatively high number of practitioner sources (32). Nonetheless, some misinterpretation or bias could have been introduced. The construction of our scoring scheme is based on an interpretation and aggregation of practitioner texts in a qualitative, empirical study [8] While precise decision drivers and impacts have been identified in the empirical study and followed by us in our scheme, an exact mapping e.g. to crisp numerical assessments would have introduced a significant threat of misinterpretations. In contrast, the ordinal scale allowed us to convert qualitative practitioner assessments to numerical data, considerably lowering this threat.

Ordinal scales are often used to reflect qualitative judgments. The threat to validity remains that some interpretations may not reflect the practitioner's judgments. This threat is mitigated by the fact that our study presents a strategy for automation rather than an empirical judgment. If others perceive any practitioner's decisions differently, it is simple to update the source code. As we provide all artifacts (code, data set) as open access artifacts to enable reproducibility, such calibration of our approach can be easily performed. The specific findings of the system evaluation would vary, but the automation technique would remain the same.

Regarding the validity of the outputs of the parser-based detectors, there is a threat that they might contain defects. To counteract this issue, we have included an empirical validation effort to ensure that the decision options are legitimate.

For the internal validity, we minimized the researcher bias by doing several independent cross-checks of our findings within the author team. Both authors independently reviewed the whole data in this study three times throughout the writing

process to prevent errors in the early stages of the research. We also offered all artifacts (code, data set) as open access artifacts to facilitate reproducibility, allowing for independent replication of our tools and evaluations.

X. Conclusion

The Link Mapping Decision (LMD) is one of the architectural design decisions in the context of the interrelation between API and domain models. It is strongly associated with API endpoints. An API description contains mainly information on API endpoints. For this reason, in this paper, we investigated how far it is possible to automatically assess conformance to LMD solely based on OpenAPI code, which has not been pursued before. To address RQ1, we developed the OpenAPI parser and assessor tools to support the following manual tasks: parsing OpenAPI descriptions, identifying the decision options, and assessing conformance. To address RQ2, we conducted empirical validations. For RQ2a, our accuracy scores towards the identification of decision options are as follows: Precision of 94.79 %, recall of 100 %, and F1-measure of 97.22 %. For RQ2b, we compared our automated approach's outcomes to the ground truth. We discovered that our results were on matching about 95.12 % correctly. This indicates that full automation of conformance assessment is indeed possible. In our future work, we plan to support other ADDs related to the interrelation between API and DDD. Our approach is potentially applicable to a variety of other model-based techniques, too. In addition, it is possible to implement our approach to other fields of interest, such as applying it to the machine-readable document format in the machine learning field or in the security and privacy field.

Acknowledgments

This work was supported by FWF (Austrian Science Fund) project API-ACE: I 4268.

References

[1] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 301–310, Jul. 2017.

[2] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun, "Introduction to microservice api patterns (map)," *Post-proceedings of Microservices 2017/2019*, vol. 78, no. 4, pp. 1–17, 2020.

[3] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.

[4] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun, "Microservice api patterns," https://microservice-api-patterns.org/, 2021.

[5] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Reading, MA.: Addison-Wesley, 2003.

[6] D. Quartel and M. van Sinderen, "On interoperability and conformance assessment in service composition," in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. IEEE, 2007, pp. 229–229.

[7] P. Rempel, P. Mäder, T. Kuschke, and J. Cleland-Huang, "Mind the gap: assessing the conformance of software traceability to relevant guidelines," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 943–954.

[8] A. Singjai, U. Zdun, and O. Zimmermann, "Practitioner views on the interrelation of microservice apis and domain-driven design: A grey literature study based on grounded theory," in *18th IEEE International Conference on Software Architecture (ICSA 2021)*. Washington, DC, USA: IEEE, March 2021, pp. –.

[9] A. Singjai and U. Zdun, "Conformance assessment of architectural design decisions on api endpoint designs derived from domain models," *Submitted for Publication*, 2022.

[10] M. Voelter, M. Kircher, and U. Zdun, *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. Hoboken, NJ, USA: J. Wiley & Sons, 2004.

[11] A. Singjai, U. Zdun, O. Zimmermann, and C. Pautasso, "Patterns on deriving apis and api endpoints from domain model elements," in *European Conference on Pattern Languages of Programs (EuroPLoP'21)*, July 2021.

[12] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker, "Interface evolution patterns: Balancing compatibility and extensibility across service life cycles," in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, ser. EuroPLop '19. New York, NY, USA: Association for Computing Machinery, 2019.

[13] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and software technology*, vol. 64, pp. 1–18, 2015.

[14] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," Technical report, Ver. 2.3 EBSE Technical Report. EBSE, Tech. Rep., 2007.

[15] M. Stocker, O. Zimmermann, U. Zdun, D. Lübke, and C. Pautasso, "Interface quality patterns: Communicating and improving the quality of microservices apis," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, ser. EuroPLoP '18. New York, NY, USA: Association for Computing Machinery, 2018.

[16] D. Persson, "Swagger reader," https://github.com/kalaspuffar/swagger-reader, 2020.

[17] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonça, "Static architecture-conformance checking: An illustrative overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.

[18] F. Tian, P. Liang, and M. A. Babar, "Relationships between software architecture and source code in practice: An exploratory survey and interview," *Information and Software Technology*, vol. 141, p. 106705, 2022.

[19] W. Hasselbring, "Software architecture: Past, present, future," in *The Essence of Software Engineering*. Springer, Cham, 2018, pp. 169–184.

[20] U. Zdun, E. Navarro, and F. Leymann, "Ensuring and assessing architecture conformance to microservice decomposition patterns," in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 411–429.

[21] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, "Assessing architecture conformance to coupling-related patterns and practices in microservices," in *European Conference on Software Architecture*. Springer, 2020, pp. 3–20.

[22] ——, "Metrics for assessing architecture conformance to microservice architecture patterns and practices," in *International Conference on Service-Oriented Computing*. Springer, 2020, pp. 580–596.

[23] M. Athanasopoulos, K. Kontogiannis, and C. Brealey, "Towards an interpretation framework for assessing interface uniformity in rest," in *Proceedings of the Second International Workshop on RESTful Design*, 2011, pp. 47–50.

[24] S. Kapferer and O. Zimmermann, "Domain-driven architecture modeling and rapid prototyping with context mapper," in *International Conference on Model-Driven Engineering and Software Development*. Springer, 2020, pp. 250–272.

[25] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2012.

[26] T. Scheller and E. Kühn, "Automated measurement of api usability: The api concepts framework," *Information and Software Technology*, vol. 61, pp. 145–162, 2015.

[27] S. M. Sohan, C. Anslow, and F. Maurer, "Spyrest: Automated restful api documentation using an http proxy server (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 271–276.