Communication Papers of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 61–68
DOI: 10.15439/2024F3164
ISSN 2300-5963 ACSIS, Vol. 41

Towards Evolvable APIs through Ontological Analysis

Nikolas Jíša
0009-0005-1551-2740
Czech Technical University in Prague
Thákurova 9, 160 00 Prague 6, Czech Republic
Email: jisaniko@fit.cvut.cz

Robert Pergl
0000-0003-2980-4400
Czech Technical University in Prague
Thákurova 9, 160 00 Prague 6, Czech Republic
Email: robert.pergl@fit.cvut.cz

***Abstract*—In recent times, the growth of technology toward decentralized solutions and microservice architecture has made Application Programming Interfaces (APIs) crucial for connecting different parts of business software systems. Although the technologies for developing and using APIs are quite stable, the fast-changing business world demands that APIs be easy to maintain and adapt. Currently, changes in APIs made by API providers often imply required updates on the side of API consumers, which can be costly and prone to mistakes. This paper analyzes the types of changes in APIs and uses this analysis to build a detailed model that shows the relationships between API consumers and API providers. This model helps visualize these relationships and can serve as a stepping stone for further automation. As a means of possible evolvable realization, we discuss the Normalized System Theory and implementation.**

I. Introduction

OVER the past few decades, there has been an extraordinary surge in technological progress, particularly evident in the widespread incorporation of the Internet into various aspects of our daily routines, such as social networking, e-commerce and banking [1]. In [2], Kurzweil proposes an extension to Moore's Law to apply the exponential growth of hardware progress to also include software and other technological areas. One such technological area could be the Internet, which incorporates a multitude of distributed systems, where services frequently depend on one another to function effectively. Among these distributed systems, there are client-server applications that can be realized through Application Programming Interfaces (APIs). Naturally, APIs evolve and are subject to changes, which is also acknowledged by Lamonthe et al. in [3]. Changes in an API on the side of the API Provider can imply other necessary changes on the side of API consumer for given API to continue to function as expected. However, keeping up with all the changes of all consumed APIs in a system, to ensure that the expected behavior matches the actual behavior, can be challenging [3]. This further highlights the critical importance of research and inquiries aimed at optimizing API development, especially if Kurzweil's Law of Accelerating Returns is right.

The focus of this paper is the management of changes on the side of the API consumer implied by updates of the consumed API with the goal of introducing more automation and reducing the manual labor needed on the side of API consumer. We delve into an analysis of this problem and then put together categories of conditions that must be satisfied for the consumption of an API to work as expected. These categories of conditions are closely related to changes on the side of API provider, implying changes on the side of API consumer. Finally, we establish an API ontological model for both the API provider side and the API consumer side, with an emphasis put on the API changes. Then use this model to argue about API evolvability challenges and their possible addressing.

The remainder of this paper is structured as follows. Section II describes the research methodology and the research goal. Section III explains relevant terms used in this paper and section IV lists existing related research works. In section V, the relevant changes in the APIs are delineated and the ontological model based on these changes is described. Section VI discusses usability of the model. Section VII evaluates the model using the identified change drivers. Section VIII concludes this paper and also mentions some ideas on follow-up research.

II. Research Overview

A. Research Methodology

The research presented in this paper adheres to Design Science Research Methodology (DSRM) [4] comprising three interconnected cycles: **1)** Relevance Cycle, which kicks off the research by linking it to real-world needs, outlining what needs to be studied, and setting clear standards for judging the outcomes **2)** Rigor Cycle, which ensures research innovation by assembling existing knowledge as the foundation for the study **3)** Design Cycle, which involves constructing an artifact, evaluating it, and incorporating feedback. This central cycle is based on the other two.

The relevance cycle for this paper is described in section II-B. The rigor cycle is described in section IV. The design cycle and its results are described in section V and their evaluation is given in section VII.

B. Research Goal

The goal of this paper is to contribute to machine actionability (i.e. automation) of changes on the side of API consumer

©2024, PTI

Topical area: Software, System and Service Engineering

implied by updates of given consumed API as depicted in Figure 1. The research questions are as follows.

RQ1: What are the changes of APIs on the side of API providers, that might imply necessity of changes on the side of API consumers, for the calling of API methods to continue working in the desired manner?

RQ2: What ontological model could describe APIs from the point of view of these changes?

As an evaluation of the model, we represent an API using the designed API model to illustrate it and visualise the identified changes in the model.

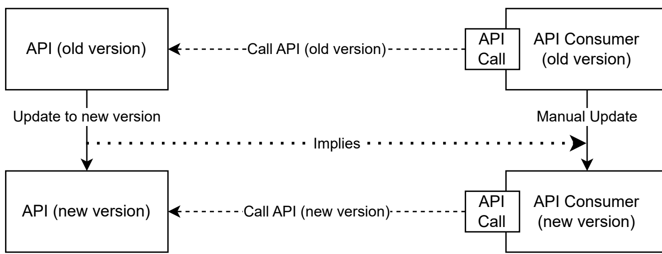

Fig. 1: API update schema

III. Theoretical Background

A. Application Programming Interface (API)

According to Reddy in [5]: "*An Application Programming Interface (API) provides an abstraction for a problem and specifies how clients should interact with software components that implement a solution to that problem.*" In this paper, we distinguish the side of API provider, which is the side that provides functionality in the form of API methods, and the side of API consumer, which is the side that consumes functionality by calls to API methods provided by the side of API provider.

B. Ontology

In this article, the term `ontology` has the meaning of `(Computational) ontology` defined in [6] as follows: "*Computational ontologies are a means to formally model the structure of a system, i.e., the relevant entities and relations that emerge from its observation, and which are useful to our purposes. An example of such a system can be a company with all its employees and their interrelationships.*"

C. Static Analysis

According to Rival et al. in [7]: "*Static analysis is an automatic technique for program-level analysis that approximates in a conservative manner semantic properties of programs before their execution.*"

D. Normalized Systems Theory

Normalized Systems Theory (NST) [8] is a theory based on fine-grained modularity with goal to make systems more evolvable and stable, specifically by elimination of Combinatorial Effects. Compliance to fine-grained modularity ensures that complex system is broken down into small components that together form the system. NST involves four theorems as follows: **T1**: Separation of Concerns: A processing function can only contain a single task in order to achieve stability **T2**: Action Version Transparency: A processing function that is called by another processing function, needs to exhibit version transparency in order to achieve stability **T3**: Data Version Transparency: A structure that is passed through the interface of a processing function needs to exhibit version transparency in order to achieve stability **T4**: Separation of States: Calling a processing function within another processing function needs to exhibit state keeping in order to achieve stability.

There exists an implementation of NST developed by NSX[1] as described in [9] that has been successfully applied to multiple real-world projects. This implementation introduces five types of elements aligned with basic software concepts as follows:

- Data Elements for data variables and structures
- Task Elements for instructions and functions
- Flow Elements for flows and orchestrations
- Connector Elements allowing input/output commands
- Trigger Elements allowing to setup triggers.

In addition to these elements, and NST Theorems, this implementation is based on code generation via expanders. Expanders are used to create instances of the elements and allow isolation of cross-cutting concerns in most cases; situations that would be difficult to cover with expanders can be handled with custom code.

E. Combinatorial Effect

According to [8], combinatorial effect is characterized by a change whose significance is influenced not only by the nature of the alteration itself, but also by the scale or scope of the system that undergoes the change.

F. FAIR Principles

`FAIR` principles are rules for scientific data management and stewardship, which were established in [10]: "*Distinct from peer initiatives that focus on the human scholar, the FAIR Principles put specific emphasis on enhancing the ability of machines to automatically find and use the data.*". Each letter of the `FAIR` acronym represents a group of principles:

F: Findability; **F1**: (Meta)data are assigned a globally unique and persistent identifier **F2**: Data are described with rich metadata (defined by **R1** below) **F3**: Metadata clearly and explicitly include the identifier of the data it describes **F4**: (Meta)data are registered or indexed in a searchable resource

A: Accessibility; **A1**: (Meta)data are retrievable by their identifier using a standardized communications protocol **A1.1**: The protocol is open, free, and universally implementable **A1.2**: The protocol allows for an authentication and authorization procedure, where necessary **A2**: Metadata are accessible, even when the data are no longer available

[1] https://normalizedsystems.org/about-us/

I: Interoperability; **I1**: (Meta)data use a formal, accessible, shared, and broadly applicable language for knowledge representation **I2**: (Meta)data use vocabularies that follow FAIR Principles **I3**: (Meta)data include qualified references to other (meta)data

R: Reusability; **R1**: Meta(data) are richly described with plurality of accurate and relevant attributes **R1.1**: (Meta)data are released with a clear and accessible data usage license **R1.2**: (Meta)data are associated with detailed provenance **R1.3**: (Meta)data meet domain-relevant community standards.

Although originally formulated for scientific data, they have been adopted to generally drive advances in the management of all types of digital objects [11]. In our work, we do not address the full scope of FAIR principles, just the parts important for machine-actionability of API evolvability.

IV. Related Work

First, we reviewed state-of-the-art of API evolvability research, and later we tried to find related ontologies to our desired API model.

A. API Evolvability Related Work

In [3] Lamonthe et al. review API Evolvability literature and, among other things, list open challenges and gaps in the research area. From these open challenges, we identified the following as being the most relevant for our paper (we also included labels from [3]):

1) EC-2 Providing a commercially viable API migration solution
2) EC-10 More tools to help with Web APIs
3) EC-15 Automatically identify factors driving API Changes
4) UC-9 Tools to help API developers deal with API migration, not just users.

For EC-2 and UC-9, there exist approaches that attempt to resolve these challenges. We identified most of the publications mentioned in this paragraph by applying snowballing [12] to [3]. In [13], Brito et al. introduce AppDiff system, which can identify breaking and non-breaking changes between two versions of a Java library based on similarity heuristics and static analysis and in [14], Dagenais et al. introduce SemDiff tool which recommends replacements for framework methods that were accessed by a client program and deleted during the evolution of the framework. However, these tools have not yet provided a commercially viable solution [3]. In [15] Ramos et al. introduce the MELT system, which can extract transformations for the API Consumer side based on the analysis of pull requests on the API Provider side based on static analysis and natural language processing of descriptions in pull requests and comments. In [16] Deshpande et al. address problem of API migration with multi-objective evolutionary algorithms without being limited to scenarios of source method getting transformed always only to single target method (one-to-one mappings), which makes this approach applicable also to scenarios of one or multiple source methods getting mapped to multiple target methods (one-to-many and many-to-many mappings). In [17], Lamothe et al. introduce system A3 for API migration of Android applications based on generation of migration code from code examples. There are also other approaches based on program synthesis that utilize examples of mapping of API calls from one version to another in order to generate transformation procedure. One such example is APIFIX tool introduced in [18], another example is ReFazer tool introduced in [19]. In [20] Beuer-Kellner et al. introduces an API Migration approach based on a service handling conversion of data structures between different versions of APIs. In [21] Huang et al. propose API Mapping approach MATL which leverages transfer learning technique to automatize API mapping without necessarily having knowledge of underlying source code of concerned APIs. One other idea is to have developers on the side of API Providers create transformation scripts for API Consumers to help update API calls from one version of API to another [3]. Similarly to our goal, the mentioned techniques deal with improving API migration. However, our focus is on providing an analytical method based on APIs modeling to identify change drivers. Our approach can then be used in combination with these techniques and other implementation technologies, such as the Normalized Systems we discuss here.

For EC-10, according to [22] Web API providers also control runtime of APIs and can do changes anytime with severe consequences to Web API consumers as opposed to Library APIs. Additionally, Web APIs often lack machine-undrestandable specifications, and data are often passed over strings. Our paper focuses on APIs in general, and the proposed API Model can later be used as a basis for contribution to Web API tooling.

For EC-15, according to Granli et al. in [23], the largest driving force for API changes is the desire for new functionality with changes occurring sporadically rather than continuously, and the Law of Conservation of Organization Stability [24] is not a considerable factor. The case study by Hou et al. in [25] shows the reasons for the changes during the evolution of the AWT[2]/Swing[3] library. The case study by Zarras et al. in [26] shows detection of evolution patterns and regularities based on Lehman's laws of software evolution. In contrast to the studies mentioned above, in this paper, we distinguish changes by the entities concerned.

B. Related Work on API Ontologies

In [27], Karavisileiou proposes a reference ontology based on OpenAPI Specification[4] for Representational State Transfer (REST) services and a procedure to convert OpenAPI Specification description into this ontology. This ontology is somewhat different from what we aim to do here, since it does not place an emphasis on combinatorial effects.

[2]https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html

[3]https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

[4]https://spec.openapis.org/oas/latest.html

In [28], Togias proposes a ontology for social network API. Although the ontology is specifically meant for social network APIs rather than APIs in general, multiple parts of the model are also applicable to APIs in general. In our ontological model, we use multiple entities that have similarities in this ontology.

In [29], Androces introduces Platform as a Service (PaaS) ontology. Although this ontology contains some entities that are applicable to our desired API ontological model (such as `Operation` or `API`), most of the entities are specific to the PaaS area.

There are also multiple API description formats, which can also be considered as types of ontologies on their own, because these formats support structured metadata that describe APIs semantically and therefore conform to ontology as defined in section III-B. Examples are OpenAPI Specification[4], API Blueprint [5] and SmartAPI[6]. Although all of these API description formats contain rich structured metadata, they do not focus on API change management.

V. Implementation and Results

A. Problem Analysis

From what Wilkinson et al. describe in [10], compliance of a system with FAIR Principles helps to introduce machine actionability in general. Therefore, compliance of APIs with FAIR Principles could be a starting point to make changes on the side of the API consumer implied by updates on the side of API provider machine actionable. However, structured rich metadata (from **F2** and **I1**) must contain relevant data for API changes on the side of API provider which impact the side of API consumer. Furthermore, the API version before an update and the API version after the update should have metadata in the same format, so that the metadata after the update can be compared with metadata before the update to get the semantic representation of the API update. This semantic representation of the API update shall then serve as an input for the automation of changes on the side of the API consumer implied by the update of API on the side of API provider.

To make APIs comply with FAIR Principles, generally, a structured API description conforming to an ontology such as one of ontologies mentioned in section IV would suffice. However, since the ontologies mentioned in section IV are not meant for managing API changes on both the API provider side and the API consumer side, we need to create our own ontology. The changes are analyzed in section V-B and based on these changes a ontological API model is created in section V-C.

B. API Change Drivers

Based on analysis of changes in multiple APIs, in order to better understand change drivers (i.e. reasons to change), we put together three categories of conditions that must be satisfied between the call to the API consumer method and the API method itself on the side of the API provider. If any condition in any of these categories changes on either side, changes on the other side may also be required for the API consumer to be able to call the given API method with the desired behavior. The conditions are as follows: **C1**: Correspondence of data transfer settings such as protocol settings and endpoint settings **C2**: Correspondence of API method signature and its meaning (on side of API provider) with API method call and its expected meaning (on side of API consumer) **C3**: Correspondence of API method behavior on side of API provider with expected behavior on side of API consumer. Examples of changes affecting given conditions are:

E1: Change to different communication protocol (affects **C1**)

E2: Change of communication protocol settings such as change of authentication, encryption, encoding, serialization ... (affects **C1**)

E3: Change of signature of API method (affects **C2**):

- Change of API method name or return type
- Creation or deletion of a API method parameter
- Change of order of API method parameters
- Change of API method parameter name or parameter type

E4: Deletion of API method (affects **C2**, **C3**)

E5: Change of meaning of API method return value or parameter values (such as change of expected units from kilometers to meters) (affects **C3**)

E6: Change of API method pre-conditions / post-conditions (affects **C3**)

E7: Change of API method behavior (affects **C3**)

E8: Change of API method mechanism resulting in drastic decrease of performance (affects **C3**)

C. API Ontological Model

We decided to create a new model from scratch inspired by these existing API ontologies rather than using any of the mentioned models (such as, for example, OpenAPI) as a starting point, because we wanted our model to focus on change drivers for modifications on API consumer side implied by modifications on API provider side and also to be more abstract than the existing API ontologies mentioned in section IV-B. We utilized Visual Paradigm Community[7] for the creation of models and diagrams.

1) API Provider Side Model: Based on our analysis of the change drivers in the previous section, we first created the API model of the side of API provider in the UML class diagram, which is shown in Figure 2. We represent the ontology in plain UML notation instead of using a formal ontology modeling framework (such as OntoUML [30]) because given the technical terms involved in modeling APIs, using such frameworks would add unnecessary complexity without providing benefits for our goal. The idea of this model is the following. **1)** `API Provider` provides `APIs` **2)** `API` contains `API Methods` **3)** `API Method` contains `API Method Signature`, `API`

[5]https://apiblueprint.org/documentation/specification.html
[6]https://smart-api.info/
[7]https://www.visual-paradigm.com/

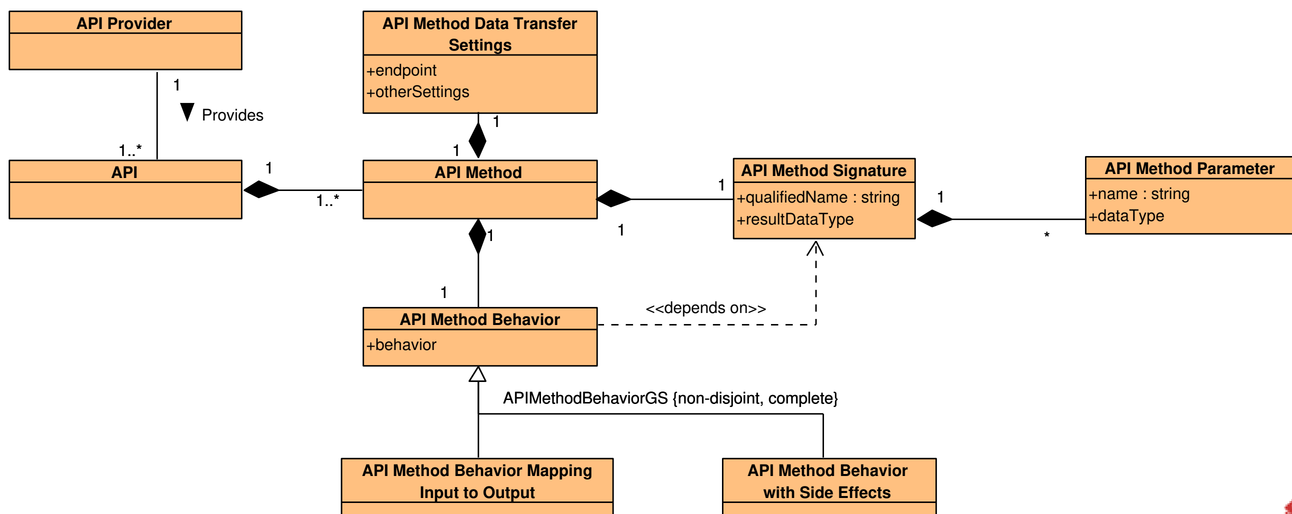

Fig. 2: The suggested API ontological model

`Method Behavior` and `API Method Data Transfer Settings` **4)** `API Method Signature` contains `name`, `resultDataType` and `API Method Parameters` **5)** `API Method Parameter` contains `name` and `dataType` **6)** `API Method Behavior` represents the behavior of the method. It can be `API Method Behavior Mapping Input to Output` and it can also be `API Method Behavior with Side Effects`; this model does not specify exactly how the `behavior` is defined - it could be, for example, natural text description, pseudocode, mapping function, or source code **7)** `API Method Data Transfer Settings` involves the configuration of communication protocol, including serialization settings, authentication settings, encryption settings, endpoint settings, ...; this model does not specify how exactly the `endpoint` and `otherSettings` are defined - `endpoint` could be, for example, defined by Uniform Resource Locator (URL) and `otherSettings` could be defined in another structured object.

2) API Consumer Side Model: Next, we extend the model by incorporating the API consumer side and highlight the dependencies between the API consumer side and the API provider side with red dashed lines in Figure 3. The idea of the entities in this model is the following: **1)** `API` provider side elements are in orange color and have the same meaning as in Figure 2 **2)** `API` consumer side elements are in red color **3)** `API Consumer` represents the entity consuming an `API` **4)** `API Method Call` represents a call of `API Method` by `API Consumer`; it consists of `name` and `API Method Call Parameter Assignments`; it has `API Method Call Data Transfer Settings` and it can also have `API Method Call Result Value Syntactic and Semantic Processing` **5)** `API Method Call Data Transfer Settings` involves configuration of communication protocol, including serialization settings, authentication settings, encryption settings, endpoint settings, ...; this model does not specify how exactly the `endpoint` and `otherSettings` are represented **6)** `API Method Call Parameter Assignment` represent what should be assigned to `API Method Parameter` identified by `name` and `dataType` in `API Method Call`; this model does not specify exactly how the `assignment` is defined - it could be, for example, natural text description, pseudocode, mapping function, or source code **7)** `API Method Call Result Value Syntactic and Semantic Processing` represents processing of the result value of the `API Method Call`; this model does not specify exactly how the `processing` is defined - it could be, for example, natural text description, pseudocode, mapping function, or source code.

As depicted in Figure 3, the API consumer side entities directly depend on the API provider side entities as follows: **1)** `API Method Call` directly depends on `API Method Signature` **2)** `API Method Call Data Transfer Settings` directly depends on `API Method Data Transfer Settings` **3)** `API Method Call Result Value Syntactic and Semantic Processing` directly depends on `API Method Behavior` and `API Method Signature` **4)** `API Method Call Parameter Assignment` directly depends on `API Method Behavior` and `API Method Parameter`.

VI. Discussion

Analysis of potential changes on the side of the API provider that could imply changes on the side of API consumer answers **RQ1** in section V-B and we also pointed out the meaning of these changes and some of their possible sources. For **RQ2**, the answer is the ontological model described in section V-C. The model serves to clarify the ontological aspects

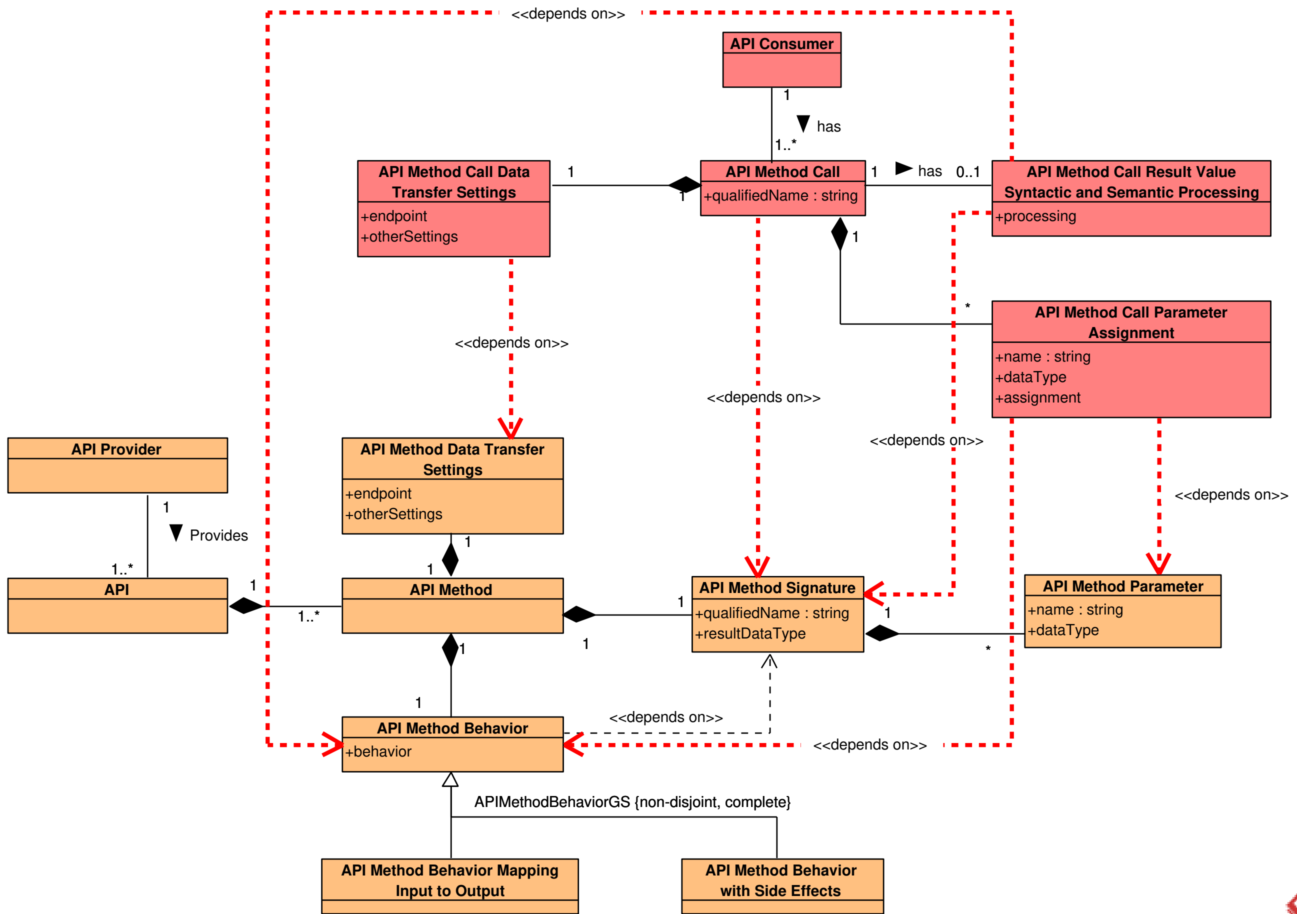

Fig. 3: API ontological model with both provider side and consumer side

of API, API provider, API consumer, and their relationships. We opt for a model that is sufficiently generic to describe most APIs without being limited by any specific domain in context of APIs such as [29] or [28]. Nevertheless, our model focuses on evolvability and change drivers, therefore, some API components (for OpenAPI for example License Object) do not have explicit semantic constructs in our model, and some other API components are defined only in an abstract manner (for OpenAPI for example Server Object, which is abstractly defined as part of API Method Data Transfer Settings in our model). Although the model is relatively simple, it can represent and be used for change analysis of any size of a real-world API (just the number of instances grows). The only current limitation is that it does not cover inter-instance dependencies, i.e. changes of one endpoint causing changes in another one. The identified change drivers could serve as a basis for applying NST, which could help mitigate combinatorial effects that cause a change made to an API system to require the same effort and scope as making the same change to a future evolved version of the API system, even if it were a thousand times larger.

To detect changes on the side of API Consumers implied by changes on the side of API Providers, artifacts such as source code, documentation artifacts, or other artifacts generated from source code or documentation artifacts could be used. One of the ways would be to compare the artifacts for a new API version with the artifacts for the previous API version. This comparison could be automatically activated as soon as a new version of API is detected and the result of this comparison could trigger a notification or even an automated script, which could update API consumption calls automatically or with possibly minimal manual intervention in the form of confirmation.

In this paper, we have considered API on an abstract level and have not covered areas in lower levels of abstraction such as security settings, licensing, or areas of concrete protocols and technologies used with APIs. Also, we have not done detailed investigation of cases when an API called (directly or indirectly) by API provider gets updated and indirectly implies changes to the side of API consumer. We have considered these cases to be the same as direct modifications on the side of API provider implying changes to the side of API consumer. We also have not covered the options concerning implementation beyond suggesting the Normalized Systems,

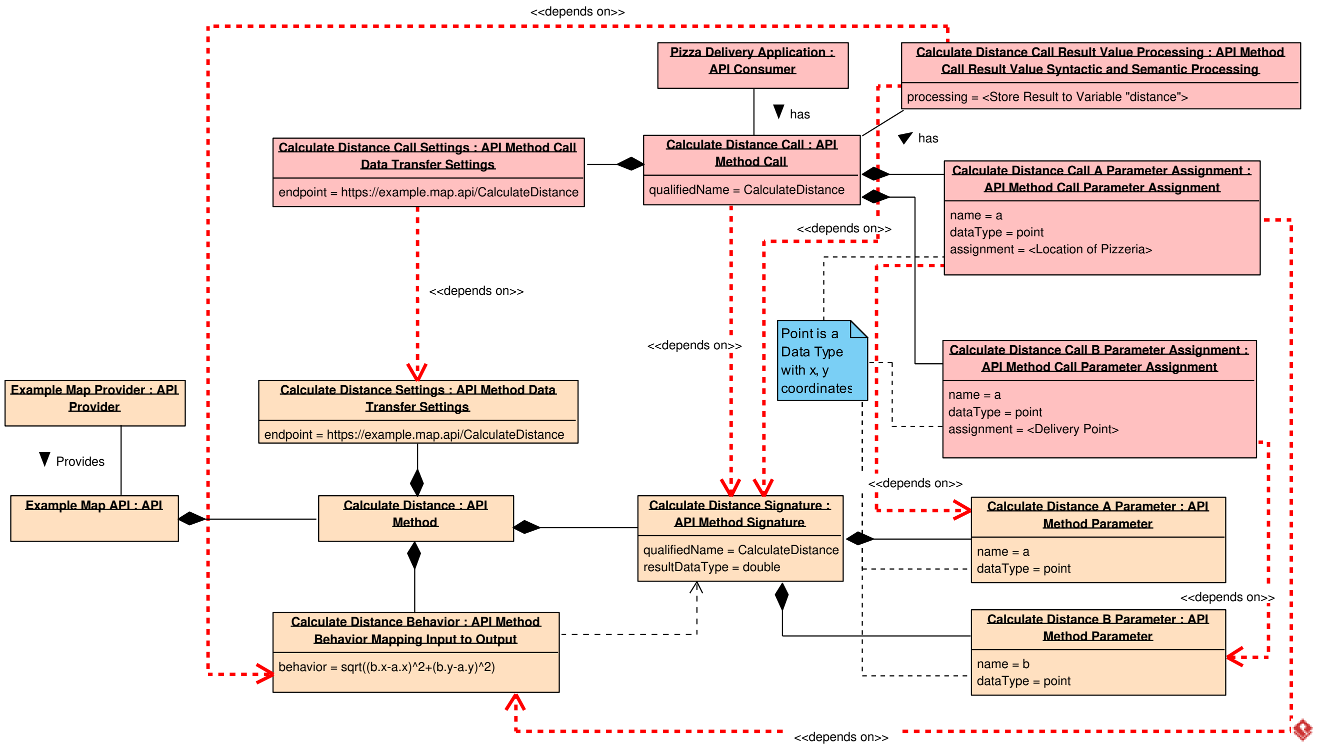

Fig. 4: Example of an API representation in API Ontological Model

which would be out of the scope of this paper.

VII. Evaluation

To verify that our model is applicable and that it can expicitly vizualize the relevant change drivers, we represent an example API with our model and later demonstrate the readability of the relevant change drivers.

Let us consider an `Example Map API` which has method `CalculateDistance` accepting parameters `a` and `b` which are of type `Point`, which has members `x` and `y` of type `double`, and the return value is of type `double`. The `Point` type could be defined in programming language C as `struct Point { double x; double y; }` and the `CalculateDistance` method would have signature `double CalculateDistance(struct Point a, struct Point b)`. Let us also consider an application for pizza delivery that consumes the API. Both the API consumer side and the API provider side could be represented in API Ontological Model as demonstrated in the UML Object Diagram[8] in Figure 4 which can be considered an instance of the model in Figure 3.

The model makes change drivers explicit to see, which we demonstrate on examples listed in section V-B as follows:

- Changes of `API Method Data Transfer Settings` in **E1** and **E2** are changes of `API Method Data Transfer Settings` on the side of API provider and imply changes of `API Method Call Data Transfer Settings` on the side of API consumer.
- Changes of `API Method Signature` in **E3** and **E4** signify changes of `API Method Signature` and `API Method Parameter` on the side of API provider and imply changes of `API Method Call`, `API Method Call Result Value Syntactic and Semantic Processing` and `API Method Call Parameter Assignment` on the side of API consumer.
- **E5**, **E6**, **E7** and **E8** are changes of `API Method Behavior` on the side of API provider and imply changes of `API Method Call Parameter Assignment` and `API Method Call Result Value Syntactic and Semantic Processing` on the side of API consumer.

[8] https://www.omg.org/spec/UML

VIII. Conclusion

The main contribution of this paper is the API ontological model, which makes it easier to see how changes on the side of the API provider affect the side of API consumer. Problem Analysis, among other things, suggests the idea of using the API ontological model as a basis for the automation of changes on the side of the API consumer implied by changes on the side of API provider. This automation could also involve an application of NST. The model could also be extended by introducing a structure to unstructured data (such as `behav-`

ior, assignment, endpoint or otherSettings) and also the idea of API SDK libraries could be applied to our model. Another interesting idea for further research would be to implement conversions between our model and other API ontology models or API description formats such as OpenAPI Specification or API Blueprint, which would make it easier to apply our model to existing APIs.

Statement on the use of AI

AI technologies (Writefull and ChatGPT) were used but only to improve the language of the paper.

REFERENCES

[1] L. Rainie and B. Wellman, *The Internet in Daily Life: The Turn to Networked Individualism*, 07 2019, pp. 27–42. ISBN 9780198843498

[2] R. Kurzweil, *The Law of Accelerating Returns*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 381–416. ISBN 978-3-662-05642-4. [Online]. Available: https://doi.org/10.1007/978-3-662-05642-4_16

[3] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of api evolution literature," *ACM Comput. Surv.*, vol. 54, no. 8, oct 2021. doi: 10.1145/3470133. [Online]. Available: https://doi.org/10.1145/3470133

[4] A. Hevner, "A three cycle view of design science research," *Scandinavian Journal of Information Systems*, vol. 19, 01 2007.

[5] M. Reddy, *API Design for C++*. Elsevier Science, 2011. ISBN 9780123850041. [Online]. Available: https://books.google.cz/books?id=IY29LylT85wC

[6] N. Guarino, D. Oberle, and S. Staab, *What Is an Ontology?* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–17. ISBN 978-3-540-92673-3. [Online]. Available: https://doi.org/10.1007/978-3-540-92673-3_0

[7] X. Rival and K. Yi, *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[8] H. Mannaert, J. Verelst, and P. De Bruyn, *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. nsi-Press powered bei Koppa, 2016. ISBN 9789077160091. [Online]. Available: https://books.google.cz/books?id=0rA_tAEACAAJ

[9] G. Oorts, K. Ahmadpour, H. Mannaert, J. Verelst, and A. Oost, "Easily evolving software using normalized system theory-a case study," *Proceedings of ICSEA*, pp. 322–327, 2014.

[10] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.

[11] K. De Smedt, D. Koureas, and P. Wittenburg, "Fair digital objects for science: From data pieces to actionable knowledge units," *Publications*, vol. 8, no. 2, 2020. doi: 10.3390/publications8020021. [Online]. Available: https://www.mdpi.com/2304-6775/8/2/21

[12] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014, pp. 1–10.

[13] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018. doi: 10.1109/SANER.2018.8330249 pp. 507–511.

[14] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *2009 IEEE 31st International Conference on Software Engineering*, 2009. doi: 10.1109/ICSE.2009.5070565 pp. 599–602.

[15] D. Ramos, H. Mitchell, I. Lynce, V. Manquinho, R. Martins, and C. L. Goues, "Melt: Mining effective lightweight transformations from pull requests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023. doi: 10.1109/ASE56229.2023.00117 pp. 1516–1528.

[16] N. Deshpande, M. W. Mkaouer, A. Ouni, and N. Sharma, "Third-party software library migration at the method-level using multi-objective evolutionary search," *Swarm and Evolutionary Computation*, vol. 84, p. 101444, 2024. doi: https://doi.org/10.1016/j.swevo.2023.101444. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S221065022300216X

[17] M. Lamothe, W. Shang, and T.-H. P. Chen, "A3: Assisting android api migrations using code examples," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 417–431, 2022. doi: 10.1109/TSE.2020.2988396

[18] X. Gao, A. Radhakrishna, G. Soares, R. Shariffdeen, S. Gulwani, and A. Roychoudhury, "Apifix: output-oriented program synthesis for combating breaking changes in libraries," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[19] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017. doi: 10.1109/ICSE.2017.44 pp. 404–415.

[20] L. Beurer-Kellner, J. von Pilgrim, C. Tsigkanos, and T. Kehrer, "A transformational approach to managing data model evolution of web services," *IEEE Transactions on Services Computing*, vol. 16, no. 1, pp. 65–79, 2023. doi: 10.1109/TSC.2022.3144613

[21] Z. Huang, J. Chen, J. Jiang, Y. Liang, H. You, and F. Li, "Mapping apis in dynamic-typed programs by leveraging transfer learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, apr 2024. doi: 10.1145/3641848. [Online]. Available: https://doi.org/10.1145/3641848

[22] E. Wittern, "Web apis - challenges, design points, and research opportunities: invited talk at the 2nd international workshop on api usage and evolution (wapi '18)," in *Proceedings of the 2nd International Workshop on API Usage and Evolution*, ser. WAPI '18. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3194793.3194801. ISBN 9781450357548 p. 18. [Online]. Available: https://doi.org/10.1145/3194793.3194801

[23] W. Granli, J. Burchell, I. Hammouda, and E. Knauss, "The driving forces of api evolution," in *Proceedings of the 14th International Workshop on Principles of Software Evolution*, ser. IWPSE 2015. New York, NY, USA: Association for Computing Machinery, 2015. doi: 10.1145/2804360.2804364. ISBN 9781450338165 p. 28–37. [Online]. Available: https://doi.org/10.1145/2804360.2804364

[24] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution-the nineties view," in *Proceedings Fourth International Software Metrics Symposium*, 1997. doi: 10.1109/METRIC.1997.637156 pp. 20–32.

[25] D. Hou and X. Yao, "Exploring the intent behind api evolution: A case study," in *2011 18th Working Conference on Reverse Engineering*, 2011. doi: 10.1109/WCRE.2011.24 pp. 131–140.

[26] A. V. Zarras, P. Vassiliadis, and I. Dinos, "Keep calm and wait for the spike! insights on the evolution of amazon services," in *ADVANCED INFORMATION SYSTEMS ENGINEERING (CAISE 2016)*, ser. Lecture Notes in Computer Science, S. Nurcan, P. Soffer, M. Bajec, and J. Eder, Eds., vol. 9694, 2016. doi: 10.1007/978-3-319-39696-5_27. ISBN 978-3-319-39696-5; 978-3-319-39695-8. ISSN 0302-9743 pp. 444–458, 28th International Conference on Advanced Information Systems Engineering (CAiSE), Ljubljana, SLOVENIA, JUN 13-17, 2016.

[27] A. Karavisileiou, N. Mainas, and E. G. Petrakis, "Ontology for openapi rest services descriptions," in *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020. doi: 10.1109/ICTAI50040.2020.00016 pp. 35–40.

[28] K. Togias and A. Kameas, "An ontology-based representation of the twitter rest api," vol. 1, 11 2012. doi: 10.1109/ICTAI.2012.85 pp. 998–1003.

[29] D. Androcec and N. Vrcek, "Platform as a service api ontology," in *PROCEEDINGS OF THE 12TH EUROPEAN CONFERENCE ON EGOVERNMENT, VOLS 1 AND 2*, M. Gasco, Ed., 2012. ISBN 978-1-908272-42-3 pp. 47–54, 12th European Conference on eGovernment (ECEG), ESADE, Inst Publ Governance & Management, Barcelona, SPAIN, JUN 14-15, 2012.

[30] G. Guizzardi, G. Wagner, J. P. Andrade Almeida, and R. S. S. Guizzardi, "Towards ontological foundations for conceptual modeling: The unified foundational ontology (ufo) story," *APPLIED ONTOLOGY*, vol. 10, no. 3-4, pp. 259–271, 2015. doi: 10.3233/AO-150157