A Precise Model for Google Cloud Platform

Stéphanie Challita, Faiez Zalila, Christophe Gourdin, Philippe Merle

A Precise Model for Google Cloud Platform

Stéphanie Challita, Faiez Zalila, Christophe Gourdin, and Philippe Merle
Inria Lille - Nord Europe & University of Lille
CRIStAL UMR CNRS 9189, France
firstname.lastname@inria.fr

***Abstract*—Today, Google Cloud Platform (GCP) is one of the leaders among cloud APIs. Although it was established only five years ago, GCP has gained notable expansion due to its suite of public cloud services that it based on a huge, solid infrastructure. GCP allows developers to use these services by accessing GCP RESTful API that is described through HTML pages on its website[1]. However, the documentation of GCP API is written in natural language (English prose) and therefore shows several drawbacks, such as *Informal Heterogeneous Documentation*, *Imprecise Types*, *Implicit Attribute Metadata*, *Hidden Links*, *Redundancy* and *Lack of Visual Support*. To avoid confusion and misunderstandings, the cloud developers obviously need a precise specification of the knowledge and activities in GCP. Therefore, this paper introduces GCP MODEL, an inferred formal model-driven specification of GCP which describes without ambiguity the resources offered by GCP. GCP MODEL is conform to the Open Cloud Computing Interface (OCCI) metamodel and is implemented based on the open source model-driven Eclipse-based OCCIWARE tool chain. Thanks to our GCP MODEL, we offer corrections to the drawbacks we identified.**

I. INTRODUCTION

During the last few years, cloud computing has become an emerging field in the IT industry. Numerous cloud providers are offering competing computation, storage, network and application hosting services, while providing coverage in several continents promising the best on-demand prices and performance. These services have heterogeneous names [1], characteristics and functionalities. In addition, cloud providers rely on technically different Application Programming Interfaces (APIs), i.e., cloud management interfaces that provide programmatic remote access to their resources. As for the semantics of these cloud APIs, they are only informally defined, i.e., described in natural language on the API website pages. The developer would not understand the exact behaviour of the provider when he/she asks for a virtual machine for example, which makes changing from one provider to another very complex and costly. However, the cloud developers need to build configurations that better fit their needs while reducing their dependence on any given provider. This is known as multi-cloud computing [2], which is the use of resources from multiple cloud providers where there is no agreement between providers. The latter is quite advantageous for cloud developers and mainly aims to better exploit offerings in the cloud market by employing a combination of cloud resources. To build multi-cloud systems, the cloud developer needs a resource-oriented model for each API in order to understand its semantics and compare cloud resources. We observe that these models do not exist but cloud APIs exhaustively describe their cloud resources and operations by providing wealthy informal documentations. Therefore, analyzing them is highly challenging due to the lack of precise semantics, which leads to confusion when comparing the cloud services and hinders multi-cloud computing. To address this problem, we have proposed in a previous work [3], FCLOUDS, which is a formal-based framework for semantic interoperability in multi-clouds. FCLOUDS contains a catalog of cloud precise models, that provide formal description of cloud APIs concepts and operations, for a better understanding of their behaviour. Having rigorously specified the structure and behaviour semantics of each cloud API, we can consequently define transformation rules between them, thus ensure their semantic interoperability. The precise models of FCLOUDS are inferred from the cloud APIs textual documentations.

Among the cloud APIs, Google Cloud Platform (GCP) is today one of the most important and growing in the cloud market. It provides developers several products to build a range of programs from simple websites to complex world-wide distributed applications. GCP offers hosting services on the same supporting infrastructure that Google uses internally for end-user products like Google Search and YouTube. This outstanding reliability results in GCP being adopted by eminent organizations such as Airbus, Coca-Cola, HTC, Spotify, etc. In addition, the number of GCP partners has also increased substantially, most notably Equinix, Intel and Red Hat.

To use cloud services, expert developers refer at first to the cloud API documentation, which is an agreement between the cloud provider and the developer on exactly how the cloud API will operate. By going through the GCP documentation, we realize that it contains wealthy information about GCP services and operations, such as the semantics of each attribute and the behaviour of each operation. However, GCP documentation is written in natural language, a.k.a. English prose, which results in human errors and/or semantic confusions. Also, the current GCP documentation lacks of visual support, hence the developer will spend considerable time before figuring out the links between GCP resources.

Our paper presents a precise model that describes GCP resources and operations, reasons about this API and provides corrections to its current drawbacks, such as *Informal Heterogeneous Documentation*, *Imprecise Types*, *Implicit Attribute Metadata*, *Hidden Links*, *Redundancy* and *Lack of Visual Support*. This is a work of reverse engineering [4], which

[1]https://cloud.google.com

is the process of extracting knowledge from a man-made documentation and re-producing it based on the extracted information. In order to formally encode the GCP API without ambiguity, we choose to infer a GCP MODEL from the GCP documentation. In fact, our approach leverages the use of Model-Driven Engineering (MDE) because it is advantageous on many levels [5], especially by providing a precise and homogeneous specification, and reducing the cost of developing complex systems. MDE allows to rise in abstraction from the implementation level to the model level, and to provide a graphical output and a formal verification of GCP structure and operations. Our GCP MODEL conforms to the OCCIWARE METAMODEL [6]. The latter is a precise metamodel of Open Cloud Computing Interface (OCCI)[2], the open cloud standard, specified by the Open Grid Forum (OGF) and defining a common interface for describing any kind of cloud computing resources.

The first contribution of this paper is an analysis of GCP documentation because it is as important as analyzing the API itself. Secondly, we propose a precise GCP MODEL that consists in a formal specification of GCP. This model, automatically built, also provides corrections for the drawbacks that we identified in GCP documentation. The remaining of this paper is structured as follows. In Section II we identify six general drawbacks of GCP documentation that motivate our work. Next, Section III describes our model-driven approach for a better description of GCP API and gives an overview of some background concepts we use in our GCP MODEL. In Section IV, we present some related work. Finally, Section V concludes the paper with future work.

II. DRAWBACKS AND MOTIVATIONS

The object of our study is the GCP documentation. GCP is a proprietary cloud platform that consists of a set of physical assets (e.g., computers and hard disk drives) and virtual resources (e.g., virtual machines, a.k.a. VMs) hosted in Google's data centers around the globe. We especially target this API because it belongs to a well-known cloud provider and because we believe it can be represented within a better formal specification.

GCP documentation is available in the form of HTML pages online. The URL[3] is the starting point of our study and the base for building our GCP MODEL. This page exhaustively lists the resources supported by the deployment manager, and provides a hyperlink to each of these resources. Normally, the developer will use the deployment manager to deploy his/her applications. The deployment manager will then provision the required resources. Therefore, we adopt this page to study the documentation of each GCP resource that could be provisioned by the developer.

Through our study of GCP, we have identified six main conceptual drawbacks/limitations on GCP documentation, which are detailed below.

[2]http://occi-wg.org/about/specification

[3]https://cloud.google.com/deployment-manager/docs/configuration/supported-resource-types

A. Informal Heterogeneous Documentation

Enforcing compliance to documentation guidelines requires specialized training and a strongly managed documentation process. However, often due to aggressive development schedules, developers neglect these extensive processes and end up writing documentations in an ad-hoc manner with some little guidance. This results in poor quality documentations that are rarely fit for any downstream software activities.

By going through the HTML pages of GCP documentation, it was not long before we realized that it has two different formats to describe the attributes of each resource (cf. Figure 1). This is an issue because it may disturb and upset the reader, i.e, the cloud developer.

Property name	Value	Description	Notes
IPv4Range	string	The range of internal addresses that are legal on this network. This range is a CIDR specification, for example: 192.168.0.0/16. Provided by the client when the network is created.	

Available at
https://cloud.google.com/compute/docs/reference/latest/networks

Vs.

Fields	
projectId	string Required. The Google Cloud Platform project ID that the cluster belongs to.

Available at
https://cloud.google.com/dataproc/docs/reference/rest/v1/projects.regions.clusters

Figure 1. Different documentation formats.

B. Imprecise Types

selfLink (1) string [Output Only] Server-defined URL for the resource.

Available at
https://cloud.google.com/compute/docs/reference/latest/targetHttpsProxies

email (2) string
The email address of the service account.
Note: This field is used in responses only. Any value specified here in a request is ignored.

Available at
https://cloud.google.com/iam/reference/rest/v1/projects.serviceAccounts

instanceClass (3) string
Instance class that is used to run this version. Valid values are:
- AutomaticScaling: F1, F2, F4, F4_1G
- ManualScaling or BasicScaling: B1, B2, B4, B8, B4_1G

Defaults to F1 for AutomaticScaling and B1 for ManualScaling or BasicScaling.

Available at
https://cloud.google.com/appengine/docs/admin-api/reference/rest/v1beta5/apps.services.versions

locations[] (4) string
The list of Google Compute Engine locations in which the cluster's nodes should be located.

Available at
https://cloud.google.com/container-engine/reference/rest/v1/projects.zones.clusters

Figure 2. Imprecise string types.

GCP documentation is represented by a huge number of descriptive tables written in natural language. Thus it is a syntactically and semantically error-prone documentation; it may contain human-errors and its static and dynamic semantics are not well-formed, i.e., does not describe without ambiguity the API and its behavior. In fact, some of the written sentences are imprecise and can be interpreted in various different ways, which can lead to confusions and misunderstandings when the user wants to provision cloud resources from GCP API. For each resource attribute, we checked the corresponding type and description to assess whether the information is accurate. Figure 2 shows that the current GCP documentation states explicitly that string types are supported. But later on, further details in the description explain how to set such strings. For example, the effective type of the attribute is a *URL* in (1), an *email address* in (2), an *enumeration* in (3), and an *array* in (4). The cloud developer may define non-valid string formats for his/her application. The bugs will be detected during the last steps of the provisioning process and fixing them becomes a tricky and time consuming task. In addition, Figure 3 shows that GCP documentation employs several ways to denote an enumeration type. Sometimes, the enumeration literals are listed in the description of the attribute, and sometimes they are retrievable from another HTML page.

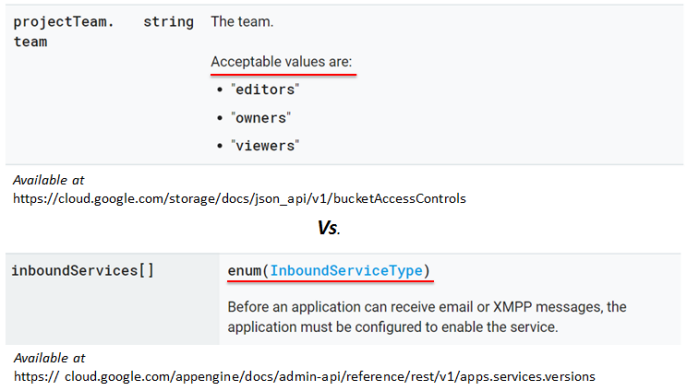


Figure 3. Informal enumeration types.

Figure 4. Error in describing the "kind" attribute.

As for Figure 4, it represents the documentation of the *kind* attribute in four different resources. We notice that (4) shows a formatting error, which induces to the fact that **GCP documentation is written by hand.**

Therefore, GCP documentation lacks of a precise and rigorous definition of its data types.

C. Implicit Attribute Metadata

We notice that **GCP documentation contains implicit information in the attribute description.** For example, it contains some information that specifies if an attribute:

- is optional or required (cf. Figure 5),

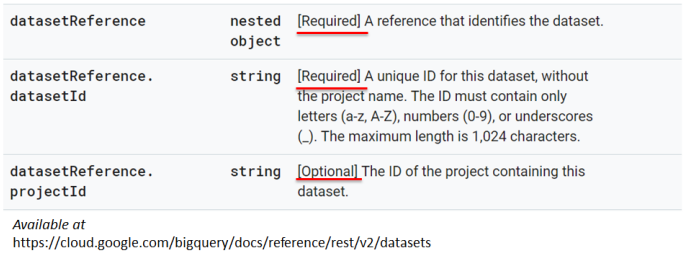


Figure 5. "Optional/Required" attribute constraint.

- is mutable or immutable (cf. Figure 6),

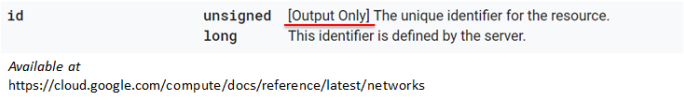


Figure 6. "Immutable attribute" constraint.

- has a default value (cf. Figure 7).

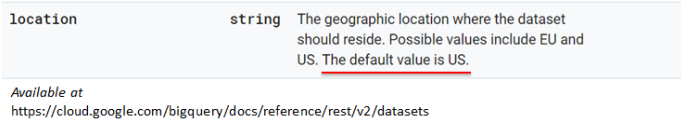
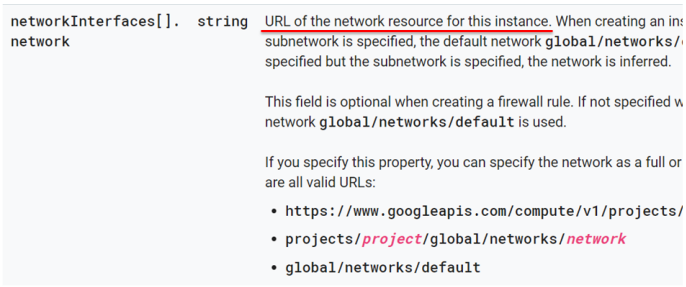


Figure 7. "Default value" constraint.

These constraints are only explained in the description of each attribute, but lacks of any verification process. The developer will not be able to ensure, before the deployment phase, that his/her code meets these constraints.

D. Hidden Links

A link is the relationship between two resource instances: a source and a target. These links are implicit in GCP documentation but they are important for proper organization of GCP resources. They are represented by a nested hierarchy, where a resource is encompassed by another resource and where an attribute defines the link between these resources, either directly or indirectly. Figure 8 shows an example of a deducible link, a.k.a. *networkInterface* because the description of this attribute is a URL pointing to the target resource, a.k.a. *network*. Therefore, we can say that *networkInterface* is a link that connects an *instance* to a *network*. If graphical support exists, this link would definitely be more explicit.



Available at
https://cloud.google.com/compute/docs/reference/latest/instances

Figure 8. Hidden link between *instance* and *network*.

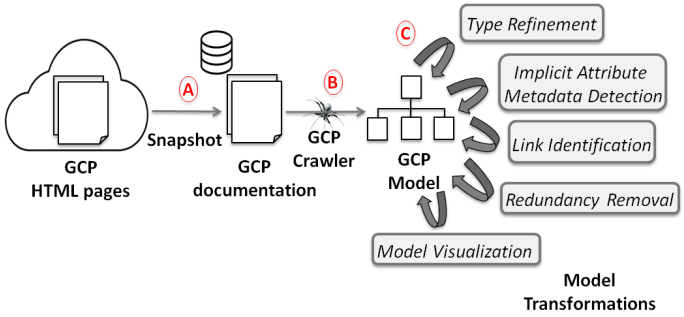


Figure 9. Architecture overview.

E. Redundancy

In addition to this, we observe from our study that **GCP documentation is redundant.** According to our observation, it contains a set of attributes and actions in common, i.e., with the same attribute name and type, and the same action name and type respectively. Among this set, we especially notice a redundancy of the attributes *name*, *id*, *kind*, *selfLink*, *description*, etc., as well as of the actions *get*, *list*, *delete*, *insert*, etc.

F. Lack of Visual Support

Finally, the information in GCP documentation is only descriptive, which involves a huge time to be properly understood and analyzed. In contrast to textual descriptions, visual diagrams help to avoid wastage of time because it easily highlights in short but catchy view the concepts of the API. Consequently, logical sequence and comparative analysis can be undertaken to enable quick understanding and attention. Cloud developers can view the graphs at a glance to understand the documentation very quickly which is more complicated through descriptive format.

Overall, these six drawbacks above are calling for more analysis of GCP documentation and for corrections. Once the development has begun, corrections can be exponentially time consuming and expensive to amend. Therefore, the cloud developer firstly needs a clear detailed specification, with no ambiguous information, in order to:

1) make the development faster and meet expectations of the cloud API,
2) avoid the different interpretations of a functionality and minimize assumptions, and,
3) help the developer to move along more smoothly with the API updates for maintainability purpose.

III. Approach

This section presents our approach that takes advantage of MDE techniques to precisely, textually and graphically, describe GCP API. In fact, MDE is emerging and emphasizing on the use of models and model transformations to raise the level of abstraction and automation in the software development.

To understand the concepts that rely under the architecture of our approach, we begin by giving an illustration of it in Figure 9. This architecture is composed of three main parts: a SNAPSHOT of GCP HTML pages, a GCP CRAWLER and a GCP MODEL increased by *Model Transformations*. Each of these three parts is detailed in the following.

A. Snapshot of GCP HTML pages

Google is the master of its cloud API and its documentation, which means that GCP engineers could update/correct GCP documentation, whenever they are requested to or they feel the urge to. But since continuously following up with GCP documentation is crippling and costly, we locally save the HTML pages of GCP documentation in order to have a snapshot of GCP API at the moment of crawling its documentation. This snapshot is built on July 27^{th}, 2017.

B. GCP Crawler

In order to study and understand GCP documentation, the main step of our approach is to extract all GCP resources, their attributes and actions and to save them in a format that is very simple and easily readable by a human. In this sense, extracting knowledge by hand from this documentation is not reliable nor representative of reality; if the documentation changes, extracted knowledge should also evolve through an automated process. Therefore, we have set up an automatic crawler to infer our GCP specification from the natural language documentation.

C. GCP Model

For a better description of the GCP resources and for reasoning over them, we propose to represent the knowledge we extracted into a model that formally specifies these resources, while providing a graphical concrete syntax and processing with transformations. This addresses the drawbacks of GCP documentation identified in Section II. Choosing the adequate metamodel when developing a model is crucial for its expressiveness [7]. In this context, a language tailored for cloud computing domain will bring us the power to easily and finely specify and validate GCP API. Therefore, we choose to adopt the OCCIWARE METAMODEL [8] because it is a precise metamodel dedicated to describe any kind of cloud resources. It is based on Open Cloud Computing Interface (OCCI) [9], an

open cloud standard defining an open interface for describing Everything as a Service (XaaS). For example, OCCI describes resources that belong to the three service layers: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). It also allows the developer to deal with a set of heterogeneous clouds. The OCCIware METAMODEL is developed in the Eclipse Modelling Framework (EMF) [10] and it defines its own data type classification system. Therefore, it easily allows to define *primitive types* such as booleans, numerics and strings, and *complex types* such as arrays, enumerations and records. In addition, thanks to its *extension* concept, OCCIware METAMODEL allows us to define a set of resource instances targeting a concrete cloud computing domain such as GCP.

In our approach, we exploit these two advantages and we build a GCP MODEL, which is an expressive model and an appropriate abstraction of the GCP API. GCP MODEL is conform to OCCIware METAMODEL, which is conform to ECORE METAMODEL. For details on the OCCIware METAMODEL, readers can refer to [6]. Thanks to OCCIware METAMODEL, our GCP MODEL provides a homogeneous specification language for GCP, which tackles the **Informal Heterogeneous Documentation** drawback, identified in Section II. It also carries out five in-place *Model Transformations* that propose corrections to face and address the other drawbacks discussed in Section II. They aim for several objectives, especially *Type Refinement*, *Implicit Attribute Metadata Detection*, *Link Identification*, *Redundancy Removal* and *Model Visualization*. We highlight in the following these correcting transformations.

- *Type Refinement* is done by adopting the data type system proposed by OCCIware METAMODEL, defining regular expressions, and using the EMF validator to check the type constraints that are attached to the attributes. For instance, among the constraints defined for the GCP MODEL, one constraint states that if the type of an attribute in the documentation is string and the description explains that this is an email address, our GCP MODEL will apply the email validation constraint for refinement purpose. This kind of information is translated into a STRINGTYPE containing the following regular expression:

  $$\hat{}[A\text{–}Z0-9._\%+-]+@[A\text{–}Z0-9.-]+\backslash\backslash.[A\text{–}Z]\{2,6\}\$$

- *Implicit Attribute Metadata Detection* to explicitly store information into additional attributes defined in the ATTRIBUTE concept of our GCP MODEL. To do so, we apply *Natural Language Processing (NLP)* [11], which is a branch of the artificial intelligence field. NLP deals with analyzing and understanding the languages that humans use naturally in order to better interface with computers. Recently, NLP techniques have made great progress and have proven to be efficient in acquiring the semantics of sentences in API documentation. Among these techniques, we use the *Word Tagging/Part-of-Speech (PoS)* [12] one. It consists in marking up a word in a text as corresponding to a particular part of speech, based on both its definition and its context. For this, we declare our pre-defined tags for some GCP specific attribute properties. Some pre-defined tags are as follows:
  - `mutable = true` if `[Input-Only]`.
  - `mutable = false` if `[Output-only]/read only`.
  - `required = true` if `[Required]`.
  - `required = false` if `[Optional]`.
- *Link Identification* to deduce logical connections between resources. Therefore, we also refer to the idea of applying NLP techniques. This time, we use *Syntactic Parsing* [13] to acquire the semantics of sentences in GCP documentation. The parse trees describe the sequential patterns that allow us to identify the semantics of a link between two resources.
- *Redundancy Removal* in order to offer the cloud developers more compact, intuitive and explicit representation of GCP resources and links. To do so, we propose to have some ABSTRACTKIND instances. An ABSTRACTKIND is an abstract class from which inherit a group of Kind instances. It allows to factorize their common attributes and actions and to reuse them. This is known as *Formal Concept Analysis (FCA)* technique [14], which is a conceptual clustering technique mainly used for producing abstract concepts and deriving implicit relationships between objects described through a set of attributes.
- *Model Visualization* for an easier analysis of the API, even if the model is not as sophisticated as the original documentation. In fact, when we visualize information, we understand, retain and interpret them better and quicker because the insights become obvious [15]. Unfortunately, as discussed in Section II, GCP does not currently provide such a visual model.

We have implemented a prototype of our approach in Java. We used jsoup library[4] for building the SNAPSHOT of GCP HTML pages and GCP CRAWLER, and the Eclipse-based OCCIware studio[5] [6] for building GCP MODEL (see AVAILABILITY section).

Once our model is built, we define GCP configurations, which represent GCP INSTANCES that conform to GCP MODEL. Then, we elaborate use cases for our model-based GCP configurations as a way of checking them. To do so, we identify the *code generation* and *model interpretation* techniques which are two of the advantages of model-driven engineering [16]. First, with the *code generation* approach, we use GCP INSTANCES to generate artifacts, such as:

- JSON files that contain the needed structured information for creating a VM for example, through GCP deployment manager,
- CURL scripts that allow us to create a VM for example via the POST action,
- Shell scripts for GCP Command Line Interface (CLI),

[4]https://jsoup.org

[5]https://github.com/occiware/OCCI-Studio

- Java or Python code for GCP Software Development Kits (SDKs) to aid in identifying bugs prior to runtime.

Second, we experiment the *model interpretation* approach, by defining the business logic of GCP CONNECTOR. The latter defines the relationship between GCP INSTANCES and their executing environment. For this, the connector provides tools that are not only used to generate the necessary artifacts corresponding to the behavior of GCP actions (create, get, insert, list, patch, update, etc.), but also to efficiently make online updates for the GCP INSTANCES elements according to the changes in the executing environment and to the models@run.time approach [17]. The generated artifacts are seamlessly executed in the executing environment thanks to MDE principles [18].

This validation process is entitled "validation by test", because it aims at verifying whether GCP INSTANCES can be executed and updated in the real world. By validating a broad spectrum of GCP INSTANCES, we validate the efficiency of our GCP MODEL.

IV. RELATED WORK

To the best of our knowledge, we provide the first work that investigates and formalizes a cloud API documentation. In [19], Petrillo et al. have focused on studying three different cloud APIs and proposed a catalog of seventy-three best practices for designing REpresentation State Transfer (REST) [20] APIs. In contrast to our work, this work is limited to analyzing the documentations of these APIs and does not propose any corrections. Two recent works were interested in studying REST APIs in general. [21] provides a framework to analyze the structure of REST APIs to study their structural characteristics, based on their Swagger documentations. [22] presents AutoREST, an approach and a prototype to automatically infer an OpenAPI specification from a REST API HTML documentation. Our work can be seen as a combination of these two previous works [21], [22], since we infer a rigorous model-driven specification from GCP HTML documentation and we provide some analysis of its corresponding API. However, in contrast to these two works, our work is specifically applied on a cloud REST API and proposes corrections to the detected deficits of its documentation. Moreover, given that it is an important but very challenging problem, analyzing natural language documents from different fields has been studied by many previous works. In [23], Zhai et al. apply NLP techniques to construct models from Javadocs in natural language. These models allow one to reason about library behaviour and were implemented to effectively model 326 Java API functions. [24] presents an approach for inferring formal specifications from API documents targeted towards code contract generation. [25] develops an API usage mining framework and its supporting tool called MAPO (Mining API usage Pattern from Open source repositories) for mining API usage patterns automatically. [26] proposes abstract models of quality use cases by inspecting information in use case text.

V. CONCLUSION AND FUTURE WORK

In this paper, we highlight six main drawbacks of GCP documentation and we argue for the need of inferring a formal specification from the current natural language documentation. To address the problem of *informal heterogeneous documentation*, we present our model-driven approach which consists in a GCP MODEL conform to OCCIWARE METAMODEL. Using our GCP CRAWLER, our model is automatically populated by the GCP resources that are documented in plain HTML pages. We also propose five *Model Transformations* to correct the remaining five drawbacks.

For future work, we want to enhance the linguistic analysis of GCP documentation for a better type refinement. We would also like to provide a GCP STUDIO, which is a dedicated model-driven environment for GCP. It will ensure a specific environment for designing configurations that conform to GCP MODEL. Finally, we aim to generate from our model, thanks to OCCIware studio facilities, a new textual documentation of GCP API. Then, we aim to strengthen our validation by conducting a survey to be taken by developers that are using GCP API. This survey will help us to verify how accurate the processed documentation is and if it actually saves their development time. Also, for ultimate measurement of our approach, we will contact Google employees who are in charge of GCP API, because we believe that their expertise is the most efficient for reviewing our work.

For long-term perspectives, we aim to analyze how suitable is the OCCIWARE METAMODEL for our purpose. Today, there exist several general modeling languages, such as UML which is widely adopted comparing to OCCIWARE METAMODEL. However, UML is generic and may not be tailored for expressing cloud computing concepts. Therefore, it will be interesting to investigate whether the additional complexity cost that OCCIWARE METAMODEL introduces for developers to learn and adapt is worth it. This study can be statistics-oriented by quantifying the number of OCCI concepts that are used in our GCP MODEL. Also, we plan to update our approach so it would automatically handle the evolution of GCP API. At the moment, this evolution is manually ensured. For automating the process, it is more practical if our crawler is less related to the structure of GCP HTML pages, because in reality the latter are constantly updated. This can be done by experimenting artificial intelligence algorithms to extract knowledge from GCP documentation, then studying whether the inferred GCP MODEL in this case will not be missing some information. Also, our model needs to incrementally detect streaming modifications, by calculating and modifying only the differences between the initially processed version and the newly modified one. Finally, we aim to extend our approach to analyze and enhance additional natural language cloud API documentations, e.g. Amazon Web Services (AWS), Microsoft Azure, Oracle, etc.

AVAILABILITY

Readers can find the snapshot of GCP documentation built on July 27^{th}, 2017, as well as our precise GCP MODEL and

its code at the following address: https://github.com/occiware/GCP-Model.

ACKNOWLEDGMENT

This work is supported by both the OCCIware[6] research and development project funded by the French Programme d'Investissements d'Avenir (PIA) and the Hauts-de-France Regional Council.

REFERENCES

[1] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Towards a REST Cloud Computing Lexicon," in *the 7th International Conference on Cloud Computing and Services Science (CLOSER)*, 2017, pp. 376–383.

[2] D. Petcu, "Multi-Cloud: Expectations and Current Approaches," in *the International workshop on Multi-cloud applications and federated clouds*. ACM, 2013, pp. 1–6.

[3] S. Challita, F. Paraiso, and P. Merle, "Towards Formal-based Semantic Interoperability in Multi-Clouds: The fclouds Framework," in *the 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 710–713.

[4] S. Rugaber and K. Stirewalt, "Model-Driven Reverse Engineering," *IEEE software*, vol. 21, no. 4, pp. 45–53, 2004.

[5] H. Bruneliere, J. Cabot, and F. Jouault, "Combining Model-Driven Engineering and Cloud Computing," in *Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA)*, 2010.

[6] F. Zalila, S. Challita, and P. Merle, "A Model-Driven Tool Chain for OCCI," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (CoopIS)*. Springer, 2017, pp. 389–409.

[7] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[8] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata, "A Precise Metamodel for Open Cloud Computing Interface," in *the 8th International Conference on Cloud Computing (CLOUD)*. IEEE, 2015, pp. 852–859.

[9] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson, "Toward an Open Cloud Standard," *IEEE Internet Computing*, vol. 16, no. 4, pp. 15–25, 2012.

[10] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[11] G. G. Chowdhury, "Natural Language Processing," *Annual review of information science and technology*, vol. 37, no. 1, pp. 51–89, 2003.

[12] D. Klein and C. D. Manning, "Accurate Unlexicalized Parsing," in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*. Association for Computational Linguistics, 2003, pp. 423–430.

[13] D. Jurafsky, "Speech and Language Processing: An Introduction to Natural Language Processing," *Computational linguistics, and speech recognition*, 2000.

[14] U. Priss, "Formal Concept Analysis in Information Science," *Arist*, vol. 40, no. 1, pp. 521–543, 2006.

[15] D. Moody, "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, 2009.

[16] D. C. Schmidt, "Model-Driven Engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[17] N. Bencomo, R. B. France, B. H. Cheng, and U. Aßmann, *Models@run.time: Foundations, Applications, and Roadmaps*. Springer, 2014, vol. 8378.

[18] F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-Driven Management of Docker Containers," in *the 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 718–725.

[19] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study," in *the International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2016, pp. 157–170.

[20] R. T. Fielding and R. N. Taylor, *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine Doctoral dissertation, 2000.

[21] F. Haupt, F. Leymann, A. Scherer, and K. Vukojevic-Haupt, "A Framework for the Structural Analysis of REST APIs," in *the International Conference on Software Architecture (ICSA)*. IEEE, 2017, pp. 55–58.

[22] H. Cao, J.-R. Falleri, and X. Blanc, "Automated Generation of REST API Specification from Plain HTML Documentation," in *the 15th International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2017, pp. 453–461.

[23] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic Model generation from Documentation for Java API Functions," in *the 38th International Conference on Software Engineering*. ACM, 2016, pp. 380–391.

[24] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring Method Specifications from Natural Language API Descriptions," in *the 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 815–825.

[25] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," *ECOOP–Object-Oriented Programming*, pp. 318–343, 2009.

[26] A. Sinha, S. M. Sutton Jr, and A. Paradkar, "Text2Test: Automated Inspection of Natural Language Use Cases," in *the 3rd International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 155–164.

[6]http://www.occiware.org