Stability Metrics for Continuous Integration of Service-Oriented Systems

Stability Metrics for Continuous Integration of Service-Oriented Systems

Dionysis Athanasopoulos[0000−0002−0720−1986] and Daniel Keenan

School of Electronics, Electrical Engineering, and Computer Science
Queen's University Belfast, Northern Ireland, United Kingdom
{D.Athanasopoulos, dkeenan21}@qub.ac.uk

Abstract. One of the key principles of the service orientation is the standardised service contract. However, the assumption that the service contract is kept unmodified during the whole life-cycle of a system is not always held. Evolution changes on the service APIs have an impact on the maintainability of their programming clients within the system making difficult the continuous integration of the services. The metrics that have currently been applied to assess the service maintainability assess the service coupling, cohesion, complexity, and granularity. Software stability can further contribute in assessing the maintainability of systems. However, it is challenging to measure the stability of service APIs without having evolved their programming clients, because it should be measured by considering the types of the evolution changes in APIs that have direct impact on the programming clients. To address this challenge, we define a set of mappings between evolved service APIs based on which the stability changes can be determined. We further specify a generic algorithm that recognises the evolution changes required on the programming clients of the evolved APIs. We finally define an initial version of a suite of metrics that estimate the stability of a service system without assuming the existence of the evolved programming clients.

Keywords: Software stability · Service API · Evolution · Continuous integration.

1 Introduction

Organizations have already migrated or developed from scratch the architecture of their software systems into service-oriented architecture (SOA) [1]. Service-orientation views systems as a composition of reusable services. Microservices currently are an increasingly popular SOA style due to the advantages microservices provide [2]. Microservices are highly cohesive and reusable services [3]. A key principle of the service orientation is the standardised service contract that consists of a set of service descriptions [1]. Each description specifies a (micro-)service from a different aspect, e.g., syntactic, semantic, behavioral, or QoS aspect [4]. The syntactic aspect specifies the syntax and the structure of the public (micro-)service API (e.g., OpenAPI[1]). We use the term API to refer to the interface of a (micro-)service.

The core implication of the principle of the standardised service contract is that if the service contract is kept unmodified, then the code of the programming clients of the

[1] https://swagger.io/specification

service will not be affected. The term of the programming client refers to the lines of code of the SOA system that invokes a (micro-)service API. However, the assumption that the service contracts are kept unmodified during the whole life-cycle of a SOA system does not always hold. Intuitively, changes in service contracts can be frequent in the case of microservices due to the low granularity and the high number of microservices that exist in a single large-scale SOA system.

To confirm our intuition, we did a preliminary assessment of the stability of the service APIs in open-source microservice systems [5] and we observed that most of the evolution changes were made in the operations of the microservice APIs. The evolution changes in the microservice APIs are significant because they have a direct impact on the programming clients of the microservices. In other words, evolution changes in the (micro-)service APIs increase the maintenance cost of SOA systems, i.e., the number of changes required in the programming clients of APIs. As an example, a change in a parameter data-type of an operation of a microservice API may trigger updates in tens of code lines that invoke the reusable operation. However, there may exist evolution changes in the (micro-)service APIs that do not affect the programming clients of the (micro-)services. For instance, a data-type modification from the `int` data-type of an input parameter of an API operation to a more general `double` data-type does not require changes on the programming client of the API. In this case, the programming client remains stable with respect to this specific evolution change in the used API.

The source code of the microservice systems that we checked is held within repositories of a collaborative development platform (e.g., GitLab[2]). We checked the evolution changes in the microservice APIs over the many pushed commits of the systems to the repositories. Collaborative platforms usually provide a level of automation in setting up and triggering continuous-integration pipelines[3]. A pipeline contains a sequence of steps (e.g., building, testing) that will be automatically executed when commits are pushed to a central repository. The continuous integration is one of the collaborative development practices followed to reduce the development and the maintenance times of software systems [6]. However, if the programming clients of an evolved API have not been updated before the API is pushed to a repository, then the execution of the continuous-integration pipeline will be broken. This frequently happens because separate (micro-)services are usually developed/maintained by separate engineers' teams.

To avoid to break a continuous-integration pipeline, developers should measure how much stable a SOA system can be before pushing the API evolution changes to a repository. However, it is challenging to calculate the system stability because it is not clear how data-type evolution changes affect the system stability. Evolution changes at the data-type level can be quite various and of a high number because API operations take as input/return as output XML/JSON schemas that generally consist of many and various data-types. On top of that, the actual value of the system stability cannot be calculated at the API evolution time because the programming clients of the evolved APIs have not been evolved yet. Thus, the following research question is raised: "How can the stability of a SOA system be estimated without having evolved the programming clients of service APIs?"

[2] https://gitlab.com
[3] docs.gitlab.com/ee/topics/autodevops

To answer this question, we searched in the literature for stability metrics. Software stability is a quality attribute that contributes to the maintainability of systems [7]. The stability of a system is defined as its resistance to the amplification of its changes (additions, modifications, deletions) [8]. To the best of our knowledge, there is no maintainability metric that assesses (micro-)service stability. Significant research has been carried out on stability in the object-oriented domain at the level of classes [9] and more recently at the lesser-explored level of packages [10]. Conceptually, the object-oriented inter-package stability is close to (micro-)service stability because a (micro-)service can be considered as a set of packages that expose a public API used by many programming clients of other (micro-)services. In other words, we do not consider the intra-package stability because it deals with the changes made in all the classes/packages of a (micro-)service that do not necessarily affect the other (micro-)services of a system. However, the package-level stability metrics proposed in [10] are coarse-grained because these metrics do not consider the kind of the data-type changes in the public APIs of packages. Given the evolution of programming clients depend on the data-type changes, the above metrics do not suffice to assess the SOA stability.

To cover this literature gap, we contribute an initial study of the API changes that affect the stability of SOA systems. We also define a set of mappings between evolved (micro-)service APIs based on which the stability changes can be determined. We further specify a generic algorithm that recognises (guided by the API mappings) the evolution changes required on the programming clients of the evolved APIs. We finally define an initial version of a suite of metrics that estimate the stability of a SOA system without assuming the existence of the evolved programming clients. The metrics use the API mappings and follow the steps of the client-evolution algorithm. The metrics can be used for both services and microservices. Thus, we use the term of the service in the remainder of the paper to refer to both services and microservices. We finally discuss the employment of our stability metrics on a real-world microservice system [5].

The rest of the paper is structured as follows. Section 2 presents the related state-of-the-art approaches and highlights the literature gaps. Section 3 defines the concepts of SOA system, (micro-)service API, and API programming client. Section 4 defines the mapping between evolved (micro-)service APIs. Section 5 defines the evolution changes on (micro-)service APIs and the evolution algorithm of programming clients. Section 6 defines our proposed suite of stability metrics. Section 7 summarizes our contribution and discusses its future directions.

2 Related Work

The existing quality metrics for (micro-)services focus on the service cohesion, coupling, complexity, and granularity [11, 12]. Further quality metrics specific for microservices have been proposed in the area of the microservice extraction from legacy systems. An example is the quality evaluation performed on extracted microservices in [13]. The authors describe a quality criterion called “functional independence”, which focuses on cohesion and coupling.

[14] highlights that the microservice quality attributes of modularity, scalability, independence, and maintainability, have been frequently discussed in the literature, with-

out though proposing specific quality metrics. The industrial research in [15] reveals that source code quality was the primary target of tools and metrics in industry, but this is missing at the level of (micro-)service architecture. [16] developed a tool which accepts microservice system source code as input, performs static analysis on the code, and computes metrics on the microservices. [17] proposes SOA maintainability metrics in terms of microservice coupling, cohesion, and complexity.

However, there is no stability metric for SOA systems. Software stability has been well explored in the domain of object-oriented software. This has resulted in metrics such as class stability metrics [7], class implementation instability [9], and more recently a suite of package metrics [10]. However, the package-level stability metrics proposed in [10] are coarse-grained because these metrics do not consider the kind of the data-type changes in the public APIs of packages. In particular, the metrics just increase by one the number of the evolution changes if a data-type has been changed, independently of the type of the change made to the data-type. Concluding, what is still missing in the literature are *inter-service stability metrics that take into account the fine-grained data-type evolution changes in the service APIs*.

3 Service API and SOA System

We define the concepts of service API, API client, and SOA system in a generic way, independently of the underlying specification language (e.g., Java, WSDL, OpenAPI).

Definition 1 (Service API). *Service API is modelled by a name and a set of operations.*

$$api := (name,\ \{op_i\})$$

Definition 2 (API Operation). *An API operation is modelled by a name and (potentially empty) input and output messages:* $op := (name,\ msg_{in},\ msg_{out})$.

Definition 3 (Operation Message). *The input/output message of an operation consists of a set of elements:* $msg := \{e_i\}$.

We define the operation message in this paper based on the leaf elements of the hierarchical structure of an XML/JSON schema, leaving as future work the consideration of the complete hierarchical structure.

Definition 4 (Message Element). *An element of a message is modelled by a tuple that consists of the name, the data-type, the min occurrence number, and the max occurrence number of the element:* $e := (name,\ type,\ min,\ max)$.

Given that the number of the instances of a leaf element appear in a schema instance equals the product of the numbers of the instances of the elements that belong to the path from the schema root to the leaf, the min/max occurrence numbers are calculated by the product of the min/max occurrence numbers of the elements of the path [18].

Definition 5 (API Programming Client). *An API programming client instantiates (assigning data values to) a number of (a part or all of) the leaf elements of the (input/output) message of an operation of a service API and finally invokes the operation.*

$$pc := \Big(api,\ op,\ \{(name,\ type,\ num,\ \{value_i\})\}\Big)$$

Definition 6 (SOA System). *A SOA system consists of a set of service APIs. Each API is associated with the set of its programming clients:* $sys := \Big\{(api,\ \{pc_i\})\Big\}$.

4 Service API Mappings

We formally define the concept of the API mapping and we leave as future work the specification of an algorithm that technically identifies these mappings (e.g., by adapting existing schema/API mapping tool [19]). The API mapping definition considers a source API (old API version) is mapped to a target API (new API version). API mappings are hierarchically structured following the hierarchical structure of service APIs.

Definition 7 (API Mapping). *An API mapping consists of the source and the target APIs, along with a set of their* $1-1$ *operation mappings:* $m_{api} := (api_s,\ api_t,\ \{m_{op}\})$.

Definition 8 (Operation Mapping). *An operation mapping consists of the source and the target operations and their* $1-1$ *message mappings,* $m_{op} := (op_s,\ op_t,\ \{m_{msg}\})$.

Definition 9 (Message Mapping). *A message mapping consists of the source and the target messages, along with the* $1-1$ *mappings between their leaf elements.*

$$m_{msg} := (msg_s,\ msg_t,\ \{m_e\})$$

Definition 10 (Element Mapping). *An element mapping consists of the source and the target elements, the absolute differences in their min/max occurrence numbers, and the amount of the information loss due to the translation of the value of the source element to conform to the data-type of the target element:* $m_e := (e_s,\ e_t,\ \Delta min,\ \Delta max,\ loss)$.

The concept of the information loss is specified in Section 5.

5 API Evolution Changes & Evolution Algorithm for API Clients

According to [20], there are the following types of evolution changes that can occur on the syntactic aspect of service APIs: i. add parameter; ii. remove parameter; iii. rename parameter; iv. change data-type of parameter; v. change min/max occurrence numbers of parameter; vi. change data-type of return value; vii. delete operation; viii. add operation; ix. rename operation; x. combine operations; xi. split operation. We focus in this paper on the first nine types of evolution changes, leaving as future work the last two types of changes. Even if there are empirical studies on how developers react to API evolution [21], these studies do not model the algorithmic steps of the client evolution. To cover this gap, we specify in Alg. 1 an initial version of the evolution algorithm for API client, taking into account the above nine types of the API evolution changes.

Alg. 1 accept as input the mappings between a source API and a target API, along with a programming client of the source API. Alg. 1 first retrieves the mapping of the source operation to the target operation. If the source operation has been deleted, then

Algorithm 1 Evolution of a programming client of an evolved service API

Input: op_s, op_t, m_{api}, pc;

1: **if** $m_{op} = (op_s,\ op_t,\ \{m_{msg}\}) \notin m_{api}$ **then** return -1;
2: **else**
3: **if** $op_s.name \neq op_t.name$ **then** pc.op.name := op_t.name;
4: **for** $pc.name \notin m_{msg}.m_e$ **do** pc := pc - (name, type, num);
5: **for** $pc.name \in m_{msg}.m_e$ **do**
6: **if** $pc.name \neq m_{msg}.m_e.e_t.name$ **then** pc.name := $m_{msg}.m_e.e_t$.name;
7: **if** $pc.type \neq m_{msg}.m_e.e_t.type$ **then**
8: **for** $pc.type.value_i$ **do** pc.type.$value_i$:= ($m_{msg}.m_e.e_t.type$) pc.type.$value_i$;
9: **if** $pc.num > m_{msg}.m_e.e_t.max$ **then** removeValues (pc);
10: **if** $pc.num < m_{msg}.m_e.e_t.max$ **then** addValues (pc);

Alg. 1 aborts without success (Alg. 1 (Step 1)). If the source operation has been renamed, then Alg. 1 renames the source operation name to the target operation name (Alg. 1 (Step 3)). Following, Alg. 1 continues with the element mappings of the messages. For each source element, Alg. 1 executes the following steps: if the source element has been deleted, then Alg. 1 deletes the element instances from the programming client (Alg. 1 (Step 4)). If the source element has been renamed, then Alg. 1 renames the source element name to the target element name (Alg. 1 (Step 6)). If the data-type of the source element has been changed, then Alg. 1 casts the data value of the element parameter to conform to the data-type of the target element (Alg. 1 (Step 8)). If the max (min) occurrence numbers of the source element is lower (resp. higher) than the instances number of the element, then Alg. 1 removes (resp. add) the extra data values of the element (Alg. 1 (Step 9 (resp. Step 10))). We assume the removal and the additional of data values are manually performed by developers.

The information loss that can happen in the casting of values of built-in data-types (e.g., converting a `double` to an `int` value) has been quantified in [22].

6 Stability Metrics for SOA System

We define the stability metrics by using the API mappings and taking into account the steps of the client-evolution algorithm. The values of each metric belong to the interval [0, 1] (the 1 value corresponds to the max stability value).

Definition 11 (System Stability). *The stability of a SOA system equals the average stability of the mapped operations of all the programming clients of the service APIs of the system:* $s_{sys} := \frac{\sum_{i=1}^{|sys.\{pc_i\}|} s_{op}(m_{op}) \mid pc_i.op \in m_{op}}{|sys.\{pc_i\}|}$.

Definition 12 (Operation Stability). *The operation stability equals the average stability of the mapped input/output operation messages:* $s_{op} := \frac{s_{msg}(m_{msg_{in}}) + s_{msg}(m_{msg_{out}})}{2}$.

Definition 13 (Message Stability). *The stability of a source message equals the average stability of the mapped message elements divided by the max number of the element between the source message and the target message:* $s_{msg} := \frac{\sum_{i=1}^{|\{m_e\}|} s_e(m_e)}{max(|msg_s|, |msg_t|)}$.

Definition 14 (Element Stability). *The element stability equals the product of the percentages of the differences in the min/max occurrence numbers and of the information loss.:* $s_e := \frac{\Delta min}{max(e_s.min, e_t.min)} * \frac{\Delta max}{max(e_s.max, e_t.max)} * loss.$

Please note we use the product operator in Def. 14 because we consider that all the terms in the calculation of the element stability are equally important.

Illustrative example. We employed our stability metrics on an open-source microservice system which is available here [5]. We indicatively chose the real-world microservice system, Apollo[4], which contains three microservices and the mean number of API operations per microservice is 65. We focused on one of those microservices, `adminservice`, and especially, on the evolution of its API across various versions of the system. Through our inspection, we observed that most of the evolution changes on the API were additions/removals of parameters and changes in the data-types of the parameters. These cases of evolution changes are taken into account by our metrics as follows. The parameter additions/removals are considered by Def. 13 because there is no mapping between the source elements and the newly added target elements, while the denominator of the message stability value equals to the max number of the elements between a source message and a target message. Finally, the data-type changes are taken into account by Def. 14 because the differences in min/max multiplicities, along with the amount of the information loss, are considered.

7 Conclusions and Future Work

To address the challenge of measuring the stability of service APIs without having first evolved their programming clients, we defined a set of mappings between evolved service APIs based on which the stability changes can be determined. We further specified a generic algorithm that recognises the evolution changes required on the programming clients of the evolved APIs. We finally defined an initial version of a suite of metrics that estimate the stability of a service system without assuming the existence of the evolved programming clients.

We introduced in this work an early version of an automated approach for measuring service stability. The road ahead includes the definition of stability metrics that consider not only the leaf elements of the hierarchical structure of an XML/JSON schema, but the complete hierarchical structure. Moreover, all the evolution changes in the interface of APIs should be taken into account, including the combination and the split of operations. Finally, a research prototype of the approach is needed that can identify the mappings and calculate the values of the stability metrics to evaluate the effectiveness of our approach on real-world (micro-)service systems.

References

1. T. Erl. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Prentice Hall, second edition edition, 2016.

[4] https://github.com/davidetaibi/Microservices_Project_List

2. S. Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.
3. D. Taibi and K. Systä. A decomposition and metric-based evaluation framework for microservices. *CoRR*, abs/1908.08513, 2019.
4. V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou. On the evolution of services. *IEEE Transactions on Software Engineering*, 38(3):609–628, 2012.
5. M. I. Rahman, S. Panichella, and D. Taibi. A curated dataset of microservices-based systems. *CoRR*, abs/1909.03249, 2019.
6. L. Zhu, L. Bass, and G. Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.
7. J. Bogner, S. Wagner, and A. Zimmermann. Towards a practical maintainability quality model for service-and microservice-based systems. In *European Conference on Software Architecture*, pages 195–198. ACM, 2017.
8. N. L. Soong. A program stability measure. In *ACM Annual conference*, pages 163–173, 1977.
9. W. Li, L. H. Etzkorn, C. G. Davis, and J. R. Talburt. An empirical study of object-oriented system evolution. *Information and Software Technology*, 42(6):373–381, 2000.
10. J. J. A. Baig, S. Mahmood, M Alshayeb, and M. Niazi. Package-level stability evaluation of object-oriented systems. *Information & Software Technology*, 116, 2019.
11. M. Perepletchikov, C. Ryan, and K. Frampton. Cohesion metrics for predicting maintainability of service-oriented software. In *International Conference on Quality Software*, pages 328–335. IEEE Computer Society, 2007.
12. M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari. Coupling metrics for predicting maintainability in service-oriented designs. In *Australian Software Engineering Conference*, pages 329–340. IEEE Computer Society, 2007.
13. W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *International Conference on Web Services*, pages 211–218. IEEE, 2018.
14. N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *International Conference on Service-Oriented Computing and Applications*, pages 44–51. IEEE Computer Society, 2016.
15. J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Assuring the evolvability of microservices: Insights into industry practices and challenges. In *International Conference on Software Maintenance and Evolution*, pages 546–556. IEEE, 2019.
16. T. Asik and Y. E. Selçuk. Policy enforcement upon software based on microservice architecture. In *International Conference on Software Engineering Research*, pages 283–287. IEEE, 2017.
17. M. Cardarelli, L. Iovino, P. Di Francesco, A. Di Salle, I. Malavolta, and P. Lago. An extensible data-driven approach for evaluating the quality of microservice architectures. In *Symposium on Applied Computing*, pages 1225–1234. ACM, 2019.
18. H. Jiang, H. Ho, L. Popa, and W.-S. Han. Mapping-driven xml transformation. In *International Conference on World Wide Web*, pages 1063–1072, 2007.
19. D. Athanasopoulos, A. V. Zarras, P. Vassiliadis, and V. Issarny. Mining service abstractions. In *International Conference on Software Engineering*, pages 944–947. ACM, 2011.
20. J. Li, Y. Xiong, X. Liu, and L. Zhang. How does web service API evolution affect clients? In *International Conference on Web Services*, pages 300–307. IEEE Computer Society, 2013.
21. A. C. Hora, R. Robbes, M. T. Valente, N. Anquetil, A. Etien, and S. Ducasse. How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal*, 26(1):161–191, 2018.
22. E. Stroulia and Y. Wang. Structural and semantic matching for assessing web-service similarity. *International Journal of Cooperative Information Systems*, pages 407–438, 2005.