Call Sequence List Distiller for Practical Stateful API Testing

Koji Yamamoto, Takao Nakagawa, Shogo Tokui, Kazuki Munakata
Fujitsu Laboratories Ltd.
{yamamoto.kouji,nakagawa-takao,tokui.shogo,munakata.kazuki}@fujitsu.com

Abstract

Necessary and sufficient combinatorial testing is important especially for continuous development to provide stateful service APIs that are invoked by an unspecified number of users. Listing API call sequences for this type of test cases is an important factor in achieving both high test coverage and short time required for test execution. This paper proposes a method to list fewer call sequences without reducing API coverage, and a method to measure the degree of adequacy of an API sequence for testing. Evaluations of more than 400 services show that the listing method reduces the number of sequences for half of the services, and that the measurement method can determine whether the reduction is possible or not for each service with high probability.

Keywords: test for microservices; call sequence listing; stateful API; API specification; API fuzzing

1. Introduction

In the development of application systems using microservices, stateful fundamental functions on remote computing nodes are combined to realize more advanced and valuable functionality. The continuous development process to provide remote side services of fundamental functions involves testing to ensure that any combination of function calls works as intended by the developers.

Each test case of the combinatorial tests for this purpose consists of three parts: a sequence of APIs to invoke functions, input parameter values of the functions, and expected output values returned by the functions. Among them, sequences ("seqs" hereinafter) of APIs are most important because the seqs determine most of the test coverage and the time required to complete the test process.

Representative previous work to list API call seqs for services is RESTler [1] to our knowledge. It lists API call seqs by appending an API to the previously listed seq that outputs values required for the API.

DOI reference number: 10.18293/SEKE2020-095

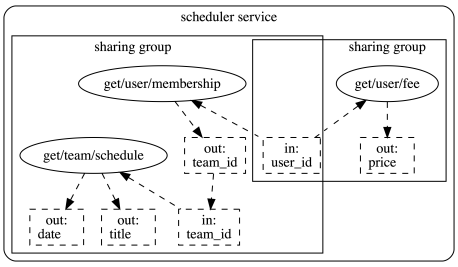

Figure 1. APIs for motivating example and value-sharing groups

From another perspective, test case enumeration pursues two types of aims. One is to find unexpected defects. The other is to ensure the functionalities are (still) as expected. The former is important for testing newly created features. The latter is crucial to continuous development of services. RESTler has achieved the former aim. So the method lists *all* the API seqs in which the value that each API takes is emitted by the predecessor APIs. However, it is necessary to reduce the number of seqs for the latter aim.

Let us see a visualized version[1] of API specification ("spec" hereinafter) in Figure 1 for a certain service. The spec includes APIs that have little to do with each other. Though some APIs in it should be called one after the other for test cases, others need not. It is hypothesized that the values handled by APIs reflect the developers' intent as to which API call should or *should not* follow a particular API call. For instance, the API get/user/membership

[1]Ovals in the figure represent APIs. Dashed rectangles stand for values emitted or consumed by APIs. Dashed arrows indicate data flow.

API spec is assumed to be written in a common format such as OpenAPI specification[2] (OAS). For example, the oval named "get/user/membership" represents the API spec in YAML style of OAS2 as follows:

```
paths:
  /user/membership:
    get: {tags: [scheduler service]
          parameters: [{name:user_id, type:string, required:true}]
          responses: {200: {
            schema: {type: array,
                     items: {type: object,
                             properties: {team_id: {type: string}},
                             required: [team_id]}}}}}
```

("API_M" for short) emits value named team_id; the API get/team/schedule ("API_S") takes the value. These indicate a call of API_M can be followed by a call of API_S. Therefore API call seq "API_M and then API_S" should be a candidate test case. The API get/user/fee ("API_F"), contrarily, does not emit values that others take, nor does it take values that others emit. So API_F should follow nothing and vice versa. To decrease in seqs, the above hypothesis can be used to avoid API seqs that are not intended by the developers.

Value-sharing groups (SGs). In order to determine the degree of adequacy of API call seqs for test cases, we propose a method to construct groups in which APIs can pass values to each other. We call the groups as *value-sharing groups*. More precisely, a value-sharing group is defined as *a minimum disjoint set of APIs that exchange values by emitting only to or receiving only from other member APIs in the same set*. In this paper, values are identified *by name*.

Value-sharing groups could be a method to measure to what extent each API call seq is adequate for a test case by counting the number of sharing groups that APIs in each seq belong to. Formally, a seq is the most adequate iff $|\{sg \in \text{SharingGroups}(spec) | sg \cap seq \neq \emptyset\}| = 1$ by using function SharingGroups in Algorithm 1 where $spec$ is a set of API specs and seq is a set of APIs contained in the seq. For example, APIs in Figure 1 are divided into two value-sharing groups drawn as two rectangles. That is, if the seq is the most adequate, the set seq of APIs in the seq holds $seq \in \mathcal{P}(\{API_M, API_S\}) \cup \mathcal{P}(\{API_F\})$ instead of $seq \in \mathcal{P}(\{API_M, API_S, API_F\})$.

This determination could help filter out API call seqs that previous work lists to reduce the time required for testing.

We preexamined API specs for 2,157 cases[2] of 410 REST services. Half of cases have more than one sharing groups. Therefore, We have developed a method to list API call seqs for testing so that each of the listed seqs is associated with one value-sharing group for almost all seqs.

This method lists API call seqs with exactly one sharing group for all the investigated cases. For 1/3 of the cases, our method lists fewer seqs than previous work. Nevertheless, API coverage by our method is equivalent to the previous work, except for one of the 2,157 cases investigated. We suppose our method reduces the number of seqs for testing without reducing test coverage.

Contributions. Contributions of this work are:

- We have developed a measurement method to decide the adequacy degree of API call seqs for testing.
- We have developed a way to list fewer API call seqs.
- We have performed quantitative evaluation of the proposed listing method using 2,157 cases of REST services. The evaluation results show our method lists fewer API call seqs for testing than previous work without reducing API coverage.

We show the proposed method and evaluation in sections 2 and 3 resp., discuss related work in 4, then conclude in 5.

2. Proposed Method

To reduce API call seqs for testing, two methods have been developed. One is to divide APIs to value-sharing groups (SGs), which appears in subsection 2.2. It is used to measure a set of API specs and an API call seqs by counting the number of associated sharing groups. Another method is to list the reduced number of API call seqs, which appears in subsection 2.3. The method is also based on the relationships between values emitted or taken by APIs.

2.1. Prerequisites

Suppose you have API specs for a service obtained by parsing the API spec file (in OpenAPI Specification [2] or other formats). Each parsed API spec corresponds to a specific API, and consists of the following information[3]:

- A set $ivals$ of tuples of the API input values. A tuple consists of a value name $name$, a boolean $reqd$ indicating that the value must be input, and a value type.
- A set $ovals$ of tuples of the API output values. The tuple type is the same as in $ivals$, but $reqd$ indicates the value must be outputted.

2.2. Value-sharing group listing function

Function SharingGroups in Algorithm 1 receives a set of API specs each of which is of type described in subsection 2.1 to output a set of SGs for the spec set.

The function creates SGs one by one. Variables $group$, ref, and $nams_N$ contain the SG being created, a set of names for values emitted or taken by at least one member API of the SG, and a set of names for values emitted or taken only by members newly added to the SG, respectively. The function attempts to select new members of the SG (ln. 5). If no member are selected, the function decides to create another SG with any API in $spec$ as an initial member (ln. 7, 11). Otherwise, the function adds selected members to the SG (ln. 11). In either case, the function adds the names for the values that the new members emit or take to

[2] In general, a service contains multiple service categories, which are identified by tags if the spec format is OAS for example. For each of the 410 services, each category identified by tag is treated as a case.

[3] A spec also contains information required to call the API. This information includes endpoint, base path, and scheme (GET, PUT, and DELETE for example) if the original API spec is in OAS.

Algorithm 1 SharingGroups

Input: A set $spec$ of API specs.
Output: A set $groups$ that stores all the sharing groups as pairs of sets. The 1st set is of APIs in a sharing group. The 2nd set is of value names that the APIs in the group take or emit.

1: $ivals \leftarrow \text{NS}(\bigcup_{api \in spec} api.ivals)$; $ovals \leftarrow \text{NS}(\bigcup_{api \in spec} api.ovals)$
2: $ungot \leftarrow ivals \setminus ovals$; $unused \leftarrow ovals \setminus ivals$
3: $groups \leftarrow \emptyset$; $nams_N \leftarrow \emptyset$
4: **while** $spec \neq \emptyset$ **do**
5: $\quad mems_N \leftarrow \{api \in spec | \text{VALNS}(api) \cap nams_N \neq \emptyset\}$
6: $\quad$ **if** $mems_N = \emptyset$ **then**
7: $\quad\quad api \leftarrow$ select an element from $spec$; $mems_N \leftarrow \{api\}$
8: $\quad\quad group \leftarrow \emptyset$; $ref \leftarrow \emptyset$ ▷ allocate new memories
9: $\quad\quad groups \leftarrow groups \cup \{\langle group, ref \rangle\}$ ▷ stores $group$ and ref as references to reflect changes after that in $groups$.
10: $\quad$ **end if**
11: $\quad group \leftarrow group \cup mems_N$
12: $\quad nams_N \leftarrow \cup_{api \in mems_N} \text{VALNS}(api) \setminus ref$
13: $\quad ref \leftarrow ref \cup nams_N$
14: $\quad nams_N \leftarrow nams_N \setminus (ungot \cup unused)$
15: $\quad spec \leftarrow sepc \setminus mems_N$
16: **end while**
17: **return** $groups$
18: **function** VALNS(api)
19: $\quad$ **return** $\text{NS}(api.ivals) \cup \text{NS}(api.ovals)$
20: **end function**

Algorithm 2 Common functions

21: **function** NS($vals$)
22: $\quad$ **return** $\{v.name | v \in vals\}$
23: **end function**

ref (ln. 13), then removes the members from $spec$ (ln. 15), and replaces $nams_N$ with a name set for the values emitted or taken only by newly added members (ln. 14).

The time complexity of Algorithm 1 is $O(S^2N)$ for S API specs and N value names because the most expensive part, ln. 5, needs $O(SN)$ at each run and is run $O(S)$ times.

Solid rectangles in Figure 1 shows the result for example.

2.3. Sequence (seq) listing algorithm

Function ListAPISeqs in Algorithm 3 takes API specs $spec$, and builds an API call seq list for testing.

First the function lists the initial API seqs (ln. 27), and stores them into the queue $todo$. Each element in the queue $todo$ is a triple of an API seq and two sets of value names that APIs in the seq take and emit resp. The function picks an API seq (ln. 29), and checks for executability by calling INVOKE[4] (ln. 30). If all the APIs in the seq have been run, the function stores it to the result list $seqlist$ (ln. 32). If the last API call has ended successfully[5], the function extends

[4]The definition of the function is omitted.
[5]For REST APIs, ListAPISeqs uses HTTP status code to judge success.

Algorithm 3 ListAPISeqs

Input: A set $spec$ of API specs, a max count N_{list} of seqs, and a max length N_{seq} of a seq.
Output: $seqlist$ that stores all listed API seqs.

24: $seqlist \leftarrow [\,]$ ▷ $seqlist$ is a list of API lists.
25: $todo \leftarrow [\,]$ ▷$todo$ is a queue for triples of an API list, and two sets of names for values taken or emitted by APIs in the list.
26: $given \leftarrow \text{NS}(\cup_{api \in spec} api.ivals) \setminus \text{NS}(\cup_{api \in spec} api.ovals)$
27: EXTEND($[\,], \emptyset, \emptyset$)
28: **while** $todo \neq [\,] \wedge |seqlist| \leq N_{list}$ **do**
29: $\quad$ dequeue $\langle seq, taken, emitted \rangle$ from $todo$
30: $\quad \langle done\_whole, last\_result \rangle \leftarrow \text{INVOKE}(seq)$
31: $\quad$ **if** $done\_whole$ **then**
32: $\quad\quad$ append seq to $seqlist$
33: $\quad\quad$ **if** $last\_result$ is successful $\wedge\ |seq| < N_{seq}$ **then**
34: $\quad\quad\quad$ EXTEND($seq, taken, emitted$)
35: $\quad\quad$ **end if**
36: $\quad$ **end if**
37: **end while**
38: **procedure** EXTEND($seq, taken, emitted$)
39: $\quad ref \leftarrow emitted \cup taken$; $feedable \leftarrow given \cup ref$
40: $\quad$ **for** each $next \in spec$ **do**
41: $\quad\quad starving \leftarrow \text{NS}(\{v \in next.ivals | v.reqd\})$
42: $\quad\quad$ **if** $starving \not\subset feedable$ **then**
43: $\quad\quad\quad$ **continue** to process rest of $next$-s
44: $\quad\quad$ **end if**
45: $\quad\quad taking \leftarrow \text{NS}(next.ivals)$
46: $\quad\quad$ **if** $seq \neq [\,] \wedge (ref \setminus given) \cap taking = \emptyset$ **then**
47: $\quad\quad\quad$ **continue** to process rest of $next$-s
48: $\quad\quad$ **end if**
49: $\quad\quad seq_N \leftarrow seq + [next]$
50: $\quad\quad taken_N \leftarrow taken \cup (feedable \cap taking)$
51: $\quad\quad emitted_N \leftarrow emitted \cup \text{NS}(next.ovals)$
52: $\quad\quad$ enqueue $\langle seq_N, taken_N, emitted_N \rangle$ to $todo$
53: $\quad$ **end for**
54: **end procedure**

it (ln. 34) for longer seqs using procedure EXTEND.

Procedure EXTEND appends a API $next$ to the specified seq seq to get a longer seq seq_N. Not all APIs are used for appending. The procedure uses the following value name sets to pick APIs to append to the seq: **(1)** $given$ – A name set of values taken by at least one APIs in $spec$ and emitted by no API. The values are treated as coming from outside the APIs in $spec$; **(2)** ref – A name set of values emitted or taken by at least one APIs in seq; **(3)** $feedable$ – A name set of values supplied by APIs in seq or externally supplied. EXTEND picks APIs that meet both of the following conditions (Note previous work employs condition **I** alone):

I All the values needed by the API are in $feedable$ (ln. 42).

II If the specified seq seq is not empty, $(ref \setminus given)$ contains at least one value taken by the API (ln. 46).

EXTEND appends each API that holds the conditions to seq

to make a new seq seq_N (ln. 49), then queues seq_N to $todo$ besides names of values taken or emitted by seq_N (ln. 52).

The time complexity of Algorithm 3 is $O(M^Q SN)$ for max API seq length Q, S API specs, N value names, and at most M members for a SG, because EXTEND, which is the most expensive and consumes $O(SN)$, is run $O(M^Q)$ times. On the other hand, the previous work needs $O(S^{Q+1}N)$. It is larger than the former complexity because $O(S) = O(GM) \geq O(M)$ where G is the number of SG.

The output seqs for APIs in Figure 1 are "API_F", "API_M", and "API_M, API_S" for example. Besides them, previous work outputs "API_F, API_M", and "API_M, API_F".

2.4. Implementation

We have implemented the functions SharingGroups and ListAPISeqs in Python3. We also have made an OAS2 parser required for the prerequisites in section 2.1. It also decomposes arrays[6] in OAS2 to obtain value names of array items, such as team_id in the motivating example API spec. The implementation includes code in which the condition **II** in section 2.3 is disabled to emulate the way of previous work like RESTler for comparison purposes. In the following, the implementation of the proposed method is called *DC* and the previous work is called *SC*.

3. Evaluation and discussion

Using the implementation above, we aim to answer the following research questions:

Q1: Does DC (proposed method) list fewer API call seqs than SC (previous work)?

Q2: Does DC decrease value-sharing groups (SGs) per seq? If so, is the decrease related to the decrease in the listed seq?

Q3: Does DC achieve the same API coverage as SC?

Q4: Is the decrease in listed call seq for a case related to the number of value-sharing groups (SGs) in that case?

To answer these fairly, we have examined *all* the API specs described in OAS 2.0 collected by APIs.guru[3][7].

This examination omits the actual API call portion of ListAPISeq[8] due to lack of access rights to the services. We listed API call seqs up to length 3. We canceled listing if the queue $todo$ was still non-empty after 2,000 seqs had been listed for each service[9]. The distribution of the examined cases with each number of SGs is shown in Figure 2.

[6]Object types are not supported yet.

[7]These specs may not have been created by the developers.

[8]It was replaced with a function that always returns $\langle true, true \rangle$.

[9]We gave up 125 of 2,282 cases in 32 of 442 services.

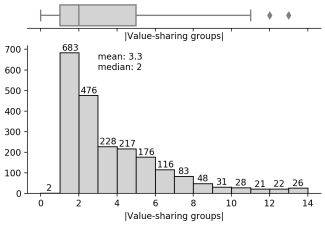

Figure 2. The examined cases distribution

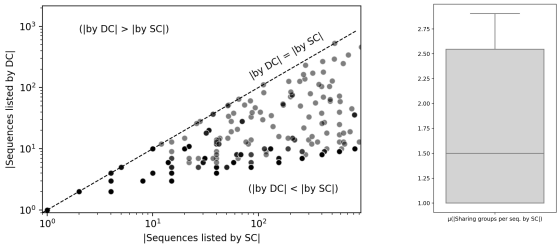

Figure 3. Plots for seqs and groups per seq

3.1. Decrease in seqs and sharing groups (Q1 & Q2)

The rows for n_s in Table 1 and the scatter plot in Figure 3 show Q1 as yes. The number n_s of seqs listed by DC is less than or equal to the number of seqs listed by SC. Each grey dot in the figure indicates how much the number of seqs for each case is reduced by DC.

The rows for n_g in the table say DC sets n_g to 1 in almost all cases. On the other hand, the box plot in Figure 3 shows the numbers of SGs per seq listed by *SC* vary from 1 to 3. Table 2 shows n_s is related to decrease of n_g. These respond affirmatively to the both questions of Q2.

Category		|Cases|
Examined cases		2,157
The number n_s of listed seqs	increased by DC	0
	equivalent	651
	decreased by DC	1,506
The number n_g of sharing groups per seq (mean value)	increased by DC	0
	equivalent (both are 1)	623
	decreased to 1 by DC	1,498
	other; no mean value	36
API coverage (The number of APIs that appear in the listed seqs)	increased by DC	0
	equivalent	2,154
	(both are 100%)	(2,033)
	(both are $< 100\%$)	(121)
	decreased by DC	1
	other; no seqs listed	2

Table 1. The numbers of cases

|Cases|	n_g decreased to 1	otherwise
n_s not decreased	0	651
n_s decreased	1498	8

(The p-value for χ^2 test is 0.0.)

Table 2. Relation of n_g and n_s

|Cases|	Cases having multiple SGs	single SG
n_s not decreased	39	612
n_s decreased	1435	71

(The p-value for χ^2 test is 0.0.)

Table 3. Relation of the number of SGs and n_s

The answer to Q1 indicates DC reduces API call seqs. The answer to Q2 says the reduction may be due to DC creating seqs containing only APIs of a single sharing group.

3.2. API coverage (Q3)

The rows for API coverage in Table 1 show that method DC holds the number of APIs that appear in the listed seqs in almost all cases while the method reduces the listed seqs.

3.3. Relationship between seq and SGs (Q4)

Table 3 shows that the fact that a case has multiple sharing groups (SGs) is related to the fact that DC lists fewer call seqs than SC. Thus, the number of SGs for a service can indicate the possibility of pruning the call seqs for a service by using the proposed method.

3.4. Threats to the validity

One threat to the validity is the services to be examined. We use API specs in APIs.guru[3] alone. The API specs may be biased while the distribution of the number of SGs in Figure 2 appears natural and an evidence of unbiased to us. Besides, we did not evaluate our method with finer grained measures (ex. code coverage) since internal information like code on the examined services is not available.

Another threat is we have not actually called APIs to examine call seqs. Even if the method is based on static analysis, the result should be confirmed by actual execution results. In particular, each seq listed by the proposed method must be checked by actual calls to see it is actually practical.

An important internal threats to the validity is that categorization by shared values may not capture the essential characteristics of API specs. There may be more intuitive and obvious factors. The scatter plot in Figure 3 implies that there can exist other drivers to control the number of seqs listed, even if the shared values is one of the drivers.

4. Related Work

Our algorithm is based on RESTler [1]. It aimed at finding unexpected results. To address another aim, seq reduction, we have to add the idea of condition **II** in section 2.3.

We suppose the proposed approach also improves methods aiming at finding unexpected results since our approach can support effective testing by reducing redundant call seqs. RESTler calls itself an API fuzzing tool. One definition of fuzzing is "the execution of the program under test (PUT) using input(s) sampled from an input space that *protrudes* the expected input space of the PUT"[4] (the emphasis is also by [4]). API seq listing without restriction does not only protrudes the input space[10] but may enlarge it explosively. Our method can control its degree.

MoonShine[5], which lists API call seqs for OS kernels, took a similar approach to ours. Its static analysis has the algorithm for cond. **I** in section 2.3, though cond. **II** is missing. As another advantage, ours depends only on API specs to support PUTs written in any programming languages.

5. Conclusion

We have developed a method to list fewer API call seqs than previous work without losing API coverage. This reduces the number of seqs in half of cases. Another developed method determines whether a seq list is reducible for each case. Nevertheless, we are afraid that these methods alone are insufficient for more practical testing of stateful service APIs. One key to improving the methods is to use attributes of values that APIs input and output more deeply (ex. on the types and the degree of necessity of the values).

References

[1] V. Atlidakis et al. RESTler: Stateful REST API fuzzing. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, 2019.

[2] OpenAPI specification. http://swagger.io/resources/open-api.

[3] APIs-guru - Wikipedia for web APIs. https://github.com/APIs-guru/openapi-directory.

[4] V. J. M. Manès et al. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering (Early Access)*, pages 1–1, 2019.

[5] Shankara Pailoor et al. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC ' 18, page 729–743, 2018.

[10] Call seqs are also inputs for combinatorial testing of services.