Issue Tracking-Based Test Data Augmentation for Web Services

JungHyun Kwon[1] Matt Staats[3] In-Young Ko[1,2] Gregg Rothermel[4]

Abstract:
Web services are widely used in software development because they offer strong advantages such as interoperability and reusability. Web application developers who use web services, however, do not own the source code of those services, which makes ensuring that service faults do not negatively impact applications a challenging, yet crucial task. Fortunately, many web service providers maintain issue tracking systems by which developers can report service issues. In this work, we present an approach for monitoring issue tracking systems and notifying service users when a new issue is registered. Our approach helps service users find test cases related to reported issues, and generates additional test cases to help users test their applications by using URIs from the issue tracking systems. We present the results of an empirical study evaluating our approach, which demonstrates that our approach generates a more targeted set of test cases for issues, and a higher rate of useful augmented test data relative to baseline approaches.

1. Introduction

Web services can be seen as building blocks in the SaaS (Software as a Service) framework. Web services are independently produced and reusable, and web service providers and service consumers usually operate independently. Thus typically, applications that access web services do not have full control of those services, rendering it difficult to determine service reliability.

One method for addressing this problem is software testing; however, in comparison to traditional software testing, there are several unique challenges in testing web services [2], [3]. First, web application developers who use web services usually cannot access the source code of the web services their applications utilize, and testing methods must therefore be black box. This is reflected in previous work on web service testing, which evaluates the reliability of web services based on service descriptions and messages that are exchanged between services [1], [7], [8], [9]. This is facilitated by RESTful (Representational State Transfer) services, which currently provide the most popular way to implement web services and maintain service API documentation written in a natural language. However, it is difficult for a non-human to read and process these service descriptions.

Second, the cost of testing web services is usually higher than the cost of testing traditional software [4]. Each test requires that a remote call be made to the service, and network latency and restrictions on the number of calls and/or the amount of time allowed to access the service (enforced by web service providers) can result in a slow and/or expensive testing process.

There has been research on testing individual web services and business processes that utilize them. However, to the best of our knowledge, there has been no research on approaches for testing applications that access RESTful web services, where faults are caused by the web services rather than by application logic.

In this work, we address this lack by providing an approach that relies on two technologies: test data augmentation and issue tracking systems. *Test data augmentation* techniques generate new test data from existing test data, with the aim of finding new variants of existing test cases that utilize program inputs in different ways, exercising different program behaviors [10] . Augmentation can increase the likelihood that a program under test works correctly, and can help engineers statistically estimate software reliability and verify fault corrections.

Issue tracking systems or issue trackers maintain a list of issues about the bugs and problems that occur in services provided. Unlike issue tracking systems for traditional software, issue tracking systems for web services use different "reproduce codes" including URIs (Uniform Resource Identifiers), XML, and JSON data. These reproduce codes can be provided as inputs to the web services. Because the reproduce codes have a formal structure, it is possible to automatically extract the codes and create new test cases based on them.

Our approach automatically analyzes URIs that have been used to reproduce faults reported in an issue tracking system, identifies relevant URIs, and creates new test data by augmenting existing test cases. The approach helps developers find new test cases related to an issue without running all test cases whenever an issue is reported. Additionally, our approach may help developers create new test cases that handle issues in advance. When a fault

[1] Div. of Web Science and Tech., KAIST, Daejeon, Korea, arunson@kaist.ac.kr, iko@kaist.ac.kr
[2] Dept. of Computer Science, KAIST, Daejeon, Korea, iko@kaist.ac.kr
[3] SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg, matthew.staats@uni.lu
[4] Dept. of Comp Science, University of Nebraska-Lincoln, Lincoln, NE, USA, grother@cse.unl.edu

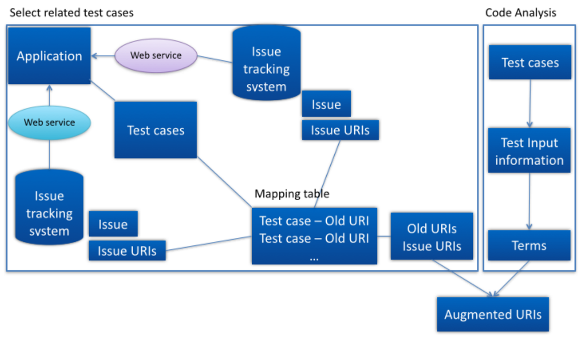

Fig. 1 Overall structure of process

is revealed by test cases, developers can provide remedies at the client level such as exception handlers and bypass code until the issue is resolved by the service provider.

To evaluate our approach, we collected issues reported in the issue tracking systems of the Google Maps geocode API, Google Maps direction API and Twitter API. Our subject applications are two Java applications. We compared the results of our approach with baseline selection and augmentation approaches. Our results indicate that our selection approach results in 12 and 5 times smaller sets of test cases and higher precision than a baseline selection approach, and our augmented approach results in a smaller but greater or equal rate of useful augmented test data than a baseline augmentation approach. Our results thus indicate that our approach can effectively produce augmented test sets.

2. Issue Tracking-Based Augmentation

Figure 1 shows the overall structure of our issue tracking-based test data augmentation process. Each web service provider manages an issue tracking system. When a new issue is registered in one of the systems, the monitoring system analyzes the issue, finds URIs and validates the URIs. URIs include operation names and parameters. In the figure, "Old URIs" are URIs generated by existing test cases for the application, and "Issue URIs" are URIs written toward the issue. Our approach measures the similarity between old URIs and issue URIs. Test cases whose URIs achieve higher similarity scores are selected as test cases related to the issue. The box on the right represents operations that analyze test case code to determine which parameters of the operations need to be augmented. This analysis finds invoked methods having arguments in the test cases, gathers identifiers and parameter names of the methods, and stores them as a list of terms. Next, the issue URIs, the related old URIs and the list of terms are used to generate new test URIs. Those URIs can be used by application developers to create additional test cases to test whether the issue affects their applications.

2.1 Monitoring Issue Tracking Systems

Application developers can obtain new issues from a mailing list that issue tracking systems manage, or by crawling the issue tracker. In this work, we considered Google Maps and Twitter APIs, each of which has associated issue trackers. We parsed reproduce code and URIs for their issues.

Issue tracking systems recommend that service consumers write "reproduce code" in their issue reports. For RESTful services, URIs can constitute reproduce code. In this work we focus on obtaining URIs automatically, so we define URI-like terms, including URIs beginning with case insensitive strings "http" or "https". Table 1 shows the numbers of issues containing URI-like terms in the issues reported until August (Google Maps geocode,Twitter) and September (Google Maps Direction), 2013. The total number of issues for these services was 105, 45, and 836, respectively. We checked to see how many service URIs were among the URI-like terms; the table shows that many issues have service URIs.

Table 1 Number of Issues Having URIs and Service URIs

	Service	Number of issues
Geocode API	Having URIs	83/105
	maps.google.com	25
	maps.googleapis.com	58
Direction API	Having URIs	31/45
	maps.google.com	9
	maps.googleapis.com	10
Twitter API	Having URIs	266/836
	api.twitter.com	73
	search.twitter.com	7
	stream.twitter.com	7

2.2 Finding Test Cases Related to Issues

We assume the common network connection methods for each language are known. For example, java.net.URL.openConnection and java.net.URL.openStream are used to invoke URIs in Java, and urllib.urlopen and urllib2.urlopen are used in Python. Next, we find the connection methods in the application source code and insert logging code behind the methods so that we can obtain the full URIs (old URIs) after running test cases.

Next, we compute similarity scores between old URIs and issue URIs. A URI consists of five components: scheme, authority (host), path, query and fragment. For RESTful services, the host name refers to a service API, and the path indicates a service operation. Further, each operation parameter is linked to a query parameter. Therefore, we measure URI similarity as follows.

(1) Check whether the authority parts of the URIs are the same.
- Check whether the two URIs belong to same service API.

(2) Measure path and query similarity.
- Identical paths indicate identical service operations. Different paths and queries may refer to different operation and method parameters.
- Use Levenshtein distance to measure similarity.
- For path similarity, measure string sequence similarity.
- For query similarity, measure string set similarity.

We use Levenshtein distance[*1] to find test cases related to an issue. Levenshtein distance calculates the minimum number of edits (insertion, deletion, substitution) needed to render two strings identical. A URI can be seen as a string so we use Levenshtein distance to measure similarity between URIs. The path part of a URI has an hierarchical structure, so different orders of paths may indicate different operations. Therefore, we measure the similarity between string sequences. For query parameters, however, the

[*1] http://en.wikipedia.org/wiki/Levenshtein_distance

order of parameters does not matter. Therefore, for these, we calculate set similarity.

After measuring the similarity of old URIs and the issue URI, the old URIs whose similarity scores are above the threshold are linked to the issue URI and the test cases invoking the old URIs are selected as test cases related to the issue.

We use the Eclipse AST parser to search for network connection methods in the applications. A regular expression is used to search for URI-like terms in the issues. Then, the detected URI-like terms are validated with the Python RFC 3987 URI validator.[*2] We measure the similarity between the old URIs and the issue URIs by using the python-levenshtein module.[*3] Next, we use methods in the levenshtein module to measure similarity between string sequences and string sets. Similarity values range from 0 to 1 and the smaller the number of edits is, the closer the similarity value is to 1. In our study, "same path" refers to the same service operation, so we set the path similarity threshold to 1. In addition, to find any URIs having at least one common parameter, we set the threshold to 0 for query similarity.

2.3 Generating Augmented Test Data

Similar to the technique proposed by McMinn et al. [6] to infer application input type, our approach analyzes the source code of the application to find test inputs. In the test code, we suggest four locations that are highly likely to include parameter names of URIs. These include (1) the identifier of the method parameter, (2) the method identifier, (3) the class identifier, and (4) the literal string containing the method arguments.

The code snippets from the four locations are used to find parameters to be augmented. We remove underscores, camel case and stop words from the snippets and determine which parameters of the URI are included in the snippets.

We use the Eclipse Abstract Syntax Tree (AST) parser to find identifiers and string literals. The parameters to be augmented are those for which identifiers and string literals contain the parameter names. For example, if a method identifier name is "setSinceId" and an issue URI's parameter name is "since_id", after converting camel case and underscores to terms separated by a space, the identifier name becomes "set since id" and the URI parameter name becomes "since id". Since "set since id" contains "since id", the augmentation technique determines that the parameter name "since_id" is the parameter to be augmented.

3. Evaluation

We conducted an empirical study to assess our test data augmentation approach. To the best of our knowledge, no comparable approach exists; we therefore compared our approach to baseline approaches described later. Our research questions were:

RQ1: Is test case selection based on mapping tables more effective than selection based on a bag of words model?

RQ2: Is test data augmentation based on code analysis more useful than augmentation based on a parameter-based approach?

3.1 Objects of Study

As objects of study, we chose two open source Java applications. The first application, Twitter4J, is a Twitter API library for Java.[*4] The second application, WorkflowProject, measures a person's social activity.[*5] Twitter4J uses the Twitter API service and WorkflowProject uses the Google Maps geocode API, the Google Maps direction API and the Facebook API. Test cases are provided with each project, and we used one test suite for Twitter4J and all of the test cases for WorkflowProject.

We used issues reported in the Twitter and Google Maps issue tracking systems (Table 1). The Google Maps direction API issues consist of 45 labeled "Defect", 44 labeled "New" and 1 labeled "FixedNotReleased". The Google Maps geocode API issues consist of 25 issues labeled "Confirmed", 72 labeled "New", 5 labeled "Acknowledged" and 3 labeled "NeedsMoreInfo". The Twitter API issues consist of 51 labeled "Acknowledged", 212 labeled "Closed", 9 labeled "InProgress" and 564 labeled "Unreviewed". Issues labeled "Not an Issue" in the Twitter issue tracker were not considered. The number of total valid URIs in the issues considered was 551 for Twitter and 241 for Google Maps.

3.2 Experiment Operation

Because there is no related work that we could find on the process of handling test data augmentation for web services, we devised two possible baselines.

For test case selection our baseline technique is based on a bag of words model. The "bag of words" model [5] is a concept used in natural language processing. We applied the model to the test cases associated with our object programs. We regarded the source code for each test case as a set of terms separated by a space. We used Apache Lucene to implement the model. The bag of words baseline removes the stop words such as Java reserved words and comments. Then, the baseline splits a test case at non-letters and converts the split terms into lowercase. The terms are stored to enable searching for relevant test cases later. Issue URIs are also converted into terms. Host, path and query parameter names for each URI are split at non-letters and are considered to be a query. The similarity measurement is based on a term frequency-inverted document frequency (tf-idf) [5]. To find relevant test cases, we set the similarity threshold to 0 and select test cases with similarity exceeding this.

As a baseline technique for test data augmentation, we created test data using parameter names common between issue URIs and old URIs without analyzing test code.

To address RQ1, we counted the number of selected test cases and measured their run times for our approach and the baseline selection technique. We compared the average precision and recall of our approach with that of the baseline to determine how many old URIs in the selected test cases used the same operations as issue URIs (precision) and how many test cases having relevant URIs were selected among all the test cases having relevant URIs (recall).

To address RQ2, first, we counted the number of runnable URIs. Runnable URIs are URIs that can receive valid responses

[*2] https://pypi.python.org/pypi/rfc3987/
[*3] https://pypi.python.org/pypi/python-Levenshtein/
[*4] https://github.com/yusuke/twitter4j
[*5] https://github.com/0925335/workflow2013

Table 2 Test Case Selection Result

	Mean number of selected tests	Mean run time	Mean Precision	Mean Recall
Twitter4j BoW	21.570	12.309	0.003	1.000
Twitter4j Mapping	1.846	4.427	1.000	1.000
Workflow BoW	9.483	13.942	0.054	1.000
Workflow Mapping	1.848	0.686	0.609	1.000

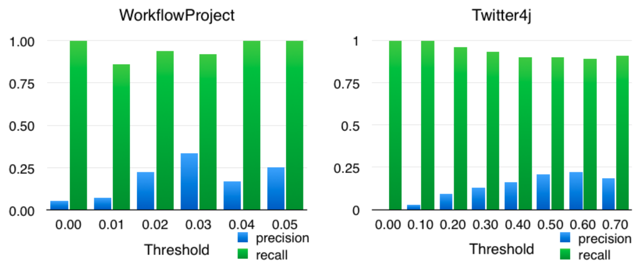

Fig. 2 Precision and recall across thresholds

Subject	# of augmented URIs		Runnable URIs		# of fixed parameters		# of handling issues	
	BoW	Map	BoW	Map	BoW	Map	BoW	Map
Twitter4J	62	62	56/62 (90.3%)	56/62 (90.3%)	0/62	0/62	8/62 (12.9%)	8/62 (12.9%)
Workflow	113	71	97/113 (85.8%)	66/71 (93.0%)	35/113 (31.0%)	0/71	79/113 (69.9%)	58/71 (81.7%)

Table 3 Test data augmentation results

from the services when used as a request. Second, we counted the number of URIs that cannot be generated from the applications. These URIs change parameter values that are fixed in the applications, so the URIs are not useful. Third, we studied the issues and counted how many augmented URIs actually handled the URI parameters in the issues.

3.3 Results

Table 2 shows the average numbers of selected test cases, average runtimes of test cases, and the average precision and average recall for every issue containing valid URIs. Our approach selected 12 and 5 times fewer test cases than the bag of words baseline for each application, respectively. Also, the runtime of our approach was 3 and 20 times faster than that of the bag of words baseline for each application, respectively. While mean recall values were the same for each approach, mean precision values for our approach were much higher than those observed for the baseline.

Low precision and recall values associated with the baseline approach may be due to our use of a low threshold. Thus, we set different threshold values and measured precision and recall again. Figure 2 shows precision and recall values across different thresholds. The x-axis corresponds to threshold and the y-axis shows precision and recall. Precision and recall of the baseline approach were lower than those of our approach at all levels.

Table 3 presents test data augmentation results. The table lists the number of augmented URIs, reusable URIs, URIs that change unchanged parameters and URIs that change parameters that an issue mentions. While the number of augmented URIs for Twitter4J using our approach was the same as that of the baseline augmentation approach, the baseline produced more augmented URIs than ours for WorkflowProject.

Next, we compared runnable URIs. For Twitter4j, the number of runnable URIs generated by the two approaches was the same. For WorkflowProject, the baseline augmentation approach generated more runnable URIs than ours, but when we consider the ratio of runnable URIs to the total number of augmented URIs, the ratio for our approach is 92.96%, which exceeds the ratio for the baseline (85.84%).

Finally, we counted the number of URIs in which the fixed parameters in the applications were changed. For both projects, our approach did not change fixed parameters, while the baseline augmentation approach changed 35 fixed parameters for WorkflowProject. We expect that these unrealistic URIs will cause developers to require longer times to create augmented test cases and the effectiveness of the test cases will be low. To measure this effect, we counted the number of augmented URIs that changed parameters related to an issue. For Twitter4J, our approach and the baseline approach recorded 12.90%. For WorkflowProject, however, our approach has 81.69% of augmented URIs handling each issue compared to 69.91% for the baseline.

4. Conclusion

Our research addresses test data augmentation for web services using URIs reported in the web services' issue tracking systems. We demonstrated that our approach selects more relevant test cases for applications using the services than the baseline selection approach in terms of precision and recall. Furthermore, our approach produces augmented test data more effectively than the baseline augmentation approach with respect to runnable URIs, changed parameters and parameter handling issues. In future work, we plan to modify our approach to reduce false positives and evaluate the approach on additional subject applications.

Acknowledgments This work was supported by the Dual Use Technology Program (UM13018RD1).

References

[1] Bai, X., Dong, W., Tsai, W.-T. and Chen, Y.: WSDL-based automatic test case generation for web services testing, *SOSE* (2005).
[2] Bozkurt, M., Harman, M. and Hassoun, Y.: Testing and verification in service-oriented architecture: A survey, *JSTVR* (2012).
[3] Canfora, G. and Penta, M.: Service-oriented architectures testing: A survey, *Software Engineering*, Springer Berlin Heidelberg (2009).
[4] Hou, S. S., Zhang, L., Xie, T. and Sun, J. S.: Quota-constrained test-case prioritization for regression testing of service-centric systems, *ICSM*, pp. 257–266 (2008).
[5] Manning, C. D., Raghavan, P. and Schütze, H.: *Introduction to Information Retrieval*, Cambridge University Press Cambridge (2008).
[6] McMinn, P., Shahbaz, M. and Stevenson, M.: Search-based test input generation for string data types using the results of web queries, *ICST* (2012).
[7] Mei, L., Zhang, Z., Chan, W. K. and Tse, T.: Test case prioritization for regression testing of service-oriented business applications, *WWW* (2009).
[8] Nguyen, C. D. and Marchetto, A.: Test case prioritization for audit testing of evolving web services using information retrieval techniques, *ICWS* (2011).
[9] Offutt, J.: Generating test cases for web services using data perturbation, *SEN* (2004).
[10] Yoo, S. and Harman, M.: Test Data Regeneration: Generating New Test Data from Existing Test Data, *JSTVR* (2010).