Metamorphic Testing for Adobe Analytics Data Collection JavaScript Library

Zhenyu Wang
Adobe
3900 Adobe Way
Lehi, Utah 84043, USA
zhenwang@adobe.com

Dave Towey
School of Computer Science
University of Nottingham Ningbo China
Ningbo 315100, China
dave.towey@nottingham.edu.cn

Zhi Quan Zhou*
Institute of Cybersecurity and Cryptology
School of Computing and Information Technology
University of Wollongong
Wollongong, NSW 2522, Australia
zhiquan@uow.edu.au

Tsong Yueh Chen
Department of Computer Science and Software Engineering
Swinburne University of Technology
Hawthorn, VIC 3122, Australia
tychen@swin.edu.au

ABSTRACT

In recent years, metamorphic testing has been successfully and systematically adopted within Adobe Systems to improve the cost effectiveness of its software process. In this industry experience report, we present a case where metamorphic testing has been applied to the Data Collection JavaScript Library of Adobe Analytics. This type of software is difficult to test using traditional approaches. The application of metamorphic testing alleviated the oracle problem, and detected real-life bugs in the system under test as well as compatibility problems between the system and its environment, namely, the Internet Explorer browser. Our results further justify the adoption of metamorphic testing as a simple yet effective approach in industrial settings.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**;

KEYWORDS

Oracle problem, metamorphic testing, Adobe Analytics, data collection, combinatorial testing

ACM Reference Format:
Zhenyu Wang, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. 2018. Metamorphic Testing for Adobe Analytics Data Collection JavaScript Library. In *MET'18: MET'18:IEEE/ACM International Workshop on Metamorphic Testing , May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3193977.3193979

*Corresponding author.

MET'18, May 27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5729-6/18/05…$15.00
https://doi.org/10.1145/3193977.3193979

1 INTRODUCTION

Adobe Analytics makes available a number of libraries [1] that can facilitate the collection of different types of data, across different platforms, such as websites and mobile apps, without the need to use the Data Insertion API (application programming interface) [2].

The libraries support different language platforms, such as JavaScript (JS) for web page data collection, a Flash library, a Java EE/SE library for collecting data from Java applications, a PHP library, and so on. The most frequently updated (active) of these libraries is one called AppMeasurement for JavaScript (AppMeasurement JS) [3].

AppMeasurement JS, which is our software under test (SUT) [4], is a compressed archive (zip) of JavaScript files, and includes different JS files for different business solutions (customers use this software development kit (SDK) library to define scenarios to use). It provides many straightforward APIs for customer usage, and can be customized in different ways to facilitate different kinds of data collection on demand.

The collected data is passed to Adobe Analytics for further analysis. Therefore, the quality of AppMeasurement JS is very important, as any fault in it may result in incorrect data being collected, causing subsequent errors in the data analytics results. This article discusses the testing of AppMeasurement JS.

The rest of this report is organized as follows: Section 2 describes the basic test flow for the SUT and the limitations of the traditional testing methods. Section 3 introduces our metamorphic testing method and describes the detected bugs. Section 4 concludes the paper.

[1] https://marketing.adobe.com/resources/help/en_US/sc/appmeasurement/release/
[2] https://marketing.adobe.com/developer/documentation/data-insertion/c-data-insertion-api
[3] More product details can be found in the following two websites: (1) https://marketing.adobe.com/resources/help/en_US/sc/appmeasurement/release/c_release_notes_mjs.html , and (2) https://marketing.adobe.com/resources/help/en_US/sc/implement/appmeasure_mjs.html .
[4] In the rest of this paper, "AppMeasurement JS" and "SUT" are used interchangably.

2 THE SUT TEST FLOW

The sequential test flow for AppMeasurement JS is as described follows:

1. Implement the customer business scenario in a web page.
2. Set up the web runtime environments.
3. Run the web page in different browsers to trigger the SUT for data collection.
4. Use a proxy server to redirect the collected data to the testing server.
5. Analyze the collected data to decide whether or not the test passes.

The SUT, AppMeasurement JS, is a JavaScript SDK library (cf. Google's Analytics library [5]) that cannot be used until the customer has configured it with appropriate values for the different variables. The first step in testing the SUT is therefore to implement the *customer business scenario* (see step 1). Example code for a customer's core JavaScript file and example customer page code are available on the Adobe website [6]. Using the code, customers can tell the SDK to track (collect) different variables (data) for analytics. AppMeasurement JS supports hundreds of variables [7], all of which can be customized by the customer. There are, for example, 250 eVar variables available for building custom reports [8]. One of these variables, *eVar1*, can be assigned an advertisement or promotion—the variable can then be used to track how many buyers add shoes to their shopping cart as a result of the advertisement or promotion. Because the variables and their combinations can be used in a very flexible way, there are infinitely many possible customer business scenarios (or "scenarios" for short).

Next, the appropriate JS scripts are loaded to set up the web environment (step 2). The various web pages are then run in the different browsers (step 3). Then, using a proxy server, the collected data are passed to the testing server (step 4), where analysis and validation of the test results can take place (step 5).

In summary, the testing involves the construction of thousands of scenarios. They are first initialized, then the library is imported, and then the tests become valid runnable web pages. The testing is conducted by running the environments (browsers). Therefore, in an abstract form, the SUT has two input parameters, namely, the scenario and the browser. If the number of scenarios to be run is T, and the number of browsers (including different browsers and different versions of the same browser) is B, then a total of $T \times B$ tests will be performed, namely: SUT(scenario$_1$, browser$_1$), SUT(scenario$_2$, browser$_1$), . . ., SUT(scenario$_T$, browser$_1$), SUT(scenario$_1$, browser$_2$), SUT(scenario$_2$, browser$_2$), . . ., SUT(scenario$_T$, browser$_2$), . . ., SUT (scenario$_T$, browser$_B$). Ideally, these tests should be run simultaneously.

In traditional testing, the test results are verified independently of one another. This means that the tester needs to verify the $T \times B$ test results separately. The testing process is depicted in Figure 1. Given that an automated test oracle for the SUT is often hard to obtain, the cost of testing is high (due to the manual labor in test result verification), and yet the effectiveness of testing in terms of failure detection can be low (due to the lack of an oracle).

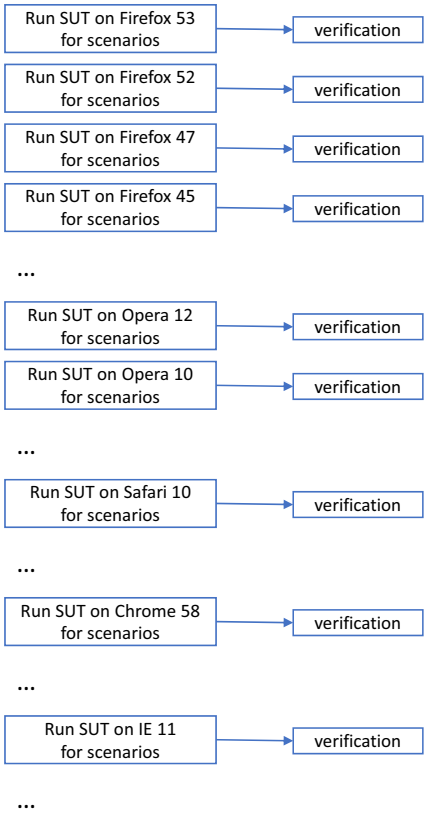

Figure 1: Traditional testing.

3 A METAMORPHIC TESTING APPROACH AND THE DETECTION OF REAL-LIFE BUGS

To improve the cost effectiveness of testing, we decided to apply *metamorphic testing* (MT) [2, 4], a testing paradigm that addresses the oracle and test automation problems [1, 3, 7]. In recent years, MT has been successfully adopted within Adobe Systems to improve the cost effectiveness of its software process (for example, see Jarman et al. [5]).

MT checks the relations among the inputs and outputs of multiple executions of the SUT. Such relations are named *metamorphic relations* (MRs), and are necessary properties of the intended software's functionality. For our SUT, AppMeasurement JS, we identified two MRs. Their definitions and implementations will be discussed in Sections 3.1 and 3.2.

3.1 MR Based on Browser Brand

The first metamorphic relation, MR1, states that, when the SUT runs on different *supported* versions of the same brand of browsers, it should return identical data (other than the browser version information) for the same scenario. The requirement "same brand of

[5] https://developers.google.com/analytics/devguides/reporting/core/v3/libraries
[6] https://marketing.adobe.com/resources/help/en_US/sc/implement/appmeasure_mjs_pagecode.html
[7] https://marketing.adobe.com/resources/help/en_US/reference/variable_definitions.html
[8] https://marketing.adobe.com/resources/help/en_US/sc/implement/eVarN.html

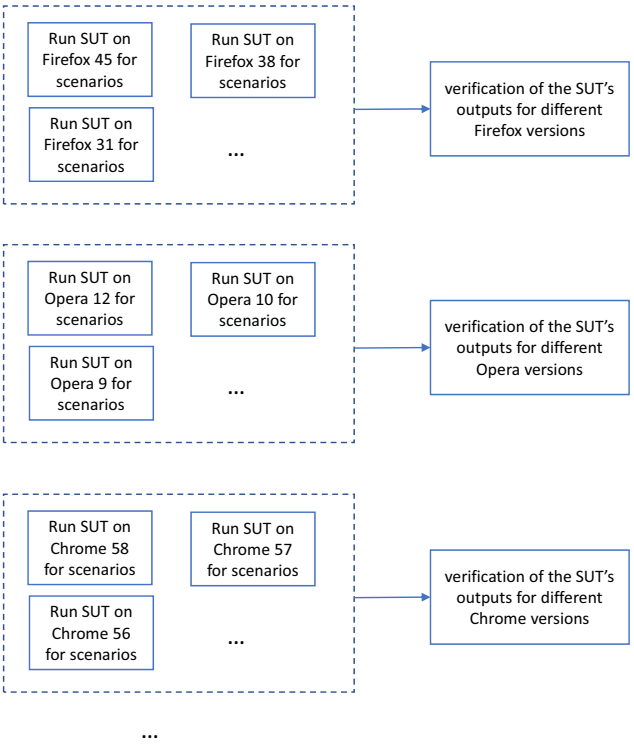

Figure 2: Metamorphic testing using MR1.
(Note: The verification is with respect to the SUT's outputs, not the browsers' outputs.)

browsers" is necessary because the SUT may return some different data fields (such as the user-agent header) when run on different brands of browsers.

Suppose the SUT supports n versions of the same brand of browser (for example, Firefox 45, Firefox 38, Firefox 31, etc), then we should put these n browser versions in the same group, as depicted in Figure 2. MR1 compares the outputs of AppMeasurement JS across the n different versions of the same brand of browsers and requires that, for the same scenario, the outputs of AppMeasurement JS should be the same (other than the browser version information). Note that *it is the outputs of the same AppMeasurement JS, not the outputs of the different browsers, that is being compared.* Therefore, what is being conducted is metamorphic testing, not *differential testing* [6]—the latter tests multiple programs using the same input and observes differences in their behavior.

Intuitively, MR1 can easily detect some types of issues such as when a browser is upgraded but the Data Collection Library has not been updated correctly. According to MR1, The test results are divided into m groups where m is the number of different browser brands. The verification of the m groups of test results can be performed in parallel and automatically. When implementing MR1, we only selected some specific data fields in the output to verify. These data fields were generated or relevant to the JS code that

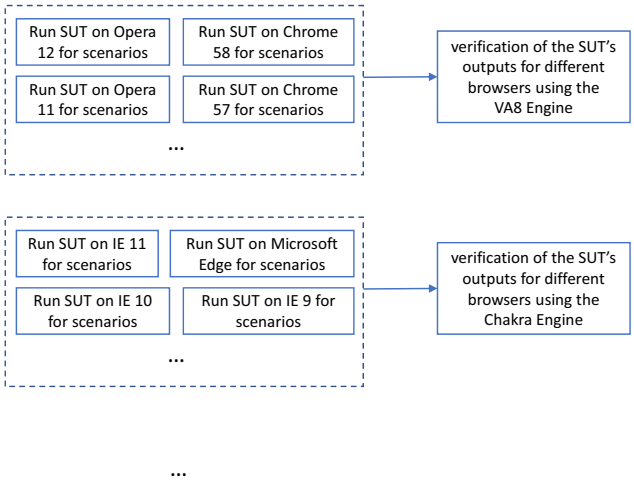

Figure 3: Metamorphic testing using MR2.

used features of the underlying browser. We found this fault-based testing strategy to be cost-effective for dealing with large amounts of output data. Furthermore, this strategy simplified the testing logic and eased the debugging process.

Using MR1, a very interesting bug in AppMeasurement JS was revealed and was recorded as **bug number AN-72873** in the Adobe Analytics Bug Management System. This is a bug in AppMeasurement JS version 1.1: When the browser is Opera version 12.6, the blank link format "about: link" triggers a data collecting event that should never happen. This is because, in Opera 12.6, the link address with a prefix "about:" is converted to an "opera:" address (in comparison, in Opera version 11 there is no such conversion). Because AppMeasurement JS 1.1 only ignores "about:" but not "opera:", it returned different outputs for test cases involving the empty link "about:" when tested on Opera 12.6 and Opera 11. MR1 was therefore violated.

3.2 MR Based on JavaScript Engine

Browsers have both render engines and JavaScript engines. Because the Data Collection Library is not involved with rendering, only the JavaScript engine is used to set up MR2. A JavaScript engine is a program that executes JavaScript code.

MR2 states that, when the SUT runs on different browsers that use the same brand of JavaScript engine (which can be of different versions), it should return identical data (other than brand and version information of the browsers and JS engines) for the same scenario.

The implementation of MR2 is very similar to that of MR1, except that the browsers may be grouped in a different way, and that different data fields in the output may be verified. The testing process is depicted in Figure 3. Note that a browser group in Figure 3 may not necessarily be a superset of a browser group in Figure 2 because different versions of the same band of browsers may use different brands of JS engines.

Intuitively, MR2 is helpful for the identification of issues related to situations where there is a change in the browser's JS engine but corresponding adjustments were not made in AppMeasurement JS to support the change.

MR1 and MR2 together mean that the SUT should have consistent behavior across those browsers that either belong to the same brand or use the same brand of JS engine. When verifying the output, MR1 implementation is focused on the output data relevant to the JS code that uses the browser's own features, whereas MR2 implementation is focused on the output data relevant to the JS code that uses the JS engine features.

Two representative issues detected by MR2 are related to Safari version 6.2 and Internet Explorer (IE) version 11. When the Data Collection Library was running in Safari 6.2, we found that the Library used an unsupported HTTP header (OPTIONS) to send data to the data collection server. This could cause data loss when collecting customer's data. Because of the detection of this issue, the Adobe development team was able to cancel the release and fix the bug in both the Library and the Server side.

The second issue was found when testing with IE 11, and was related to a limitation [9] in the Microsoft JS engine named Chakra. When running test cases in the Chakra JS engine, we found the issue of IE 11 not truncating long requests as IE 10 and IE 9 did, resulting in inaccurate data being collected. Had tests only been conducted using the traditional way (without the grouping), it would have been very difficult to identify this minor difference—but this minor difference could have a very significant impact on the data collection. This problem was detected as follows: IE 11 uses the same brand of JS engine as IE 10 and IE 9, Chakra. When the collected data (requests in URL) were too long, IE 11 did not truncate the data as IE 10 and IE 9 did (because they used different versions of the JS engine, albeit the same brand). Our data collection SUT did not expect this behavior of IE 11 and, therefore, returned different data for IE 11 as compared with IE 10 and IE 9. This was a compatibility issue between the SUT and the JS engine of IE 11. The SUT was then updated to rectify this problem. This problem was not detected when testing the SUT using MR1, because MR1 and MR2 implementations looked at different data fields in the output.

4 CONCLUSION AND FUTURE WORK

In this report, we have discussed the testing of Adobe Analytics Data Collection JavaScript Library. To test this SUT, various scenarios and a great variety of running environments (browsers with different configurations) need to be employed to generate a large number of combinations for testing. This means that a large number of test results need to be verified, but there is an oracle problem for this type of software. Conventional methods verify each output separately, which is neither efficient nor effective.

Using a metamorphic testing method, we were able to group the large number of tests into a relatively small number of groups. Within each group, the SUT's outputs are expected to be identical except for some predefined sections where the values can differ. As a preliminary study, we used two approaches to group the tests: by browser brand and by JS engine brand. Our method is very simple and intuitive, and can be fully automated and parallelized. Using this method, we have shown the detection of two real-life bugs and one compatibility issue involving the Microsoft browser IE 11.

A limitation of this study is that the two MRs are very simple and, hence, they might not be sensitive to some types of faults. Further analysis needs to be conducted to identify a more complete list of MRs to enhance the fault-detection effectiveness.

The results of this study justify the adoption of metamorphic testing in industrial settings, and provide evidence to show that even very simple metamorphic relations can detect significant real-life bugs in major commercial software. In particular, we wish to point out that both our previous study [5] and our present one have been conducted in the area of data analytics, where the former tested Adobe data analysis software and the latter tested Adobe data collection software. The findings of these two studies provide evidence of the applicability and cost effectiveness of metamorphic testing for data analytics applications.

Future work will include more systematic identification of MRs for a variety of tasks in both data analytics and other types of applications.

[9] https://support.microsoft.com/en-us/help/208427/maximum-url-length-is-2,083-characters-in-internet-explorer

ACKNOWLEDGMENT

This work was supported in part by a linkage grant of the Australian Research Council (project ID: LP160101691).

REFERENCES

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[2] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic testing: A new approach for generating next test cases.* Technical Report HKUST-CS98-01. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

[3] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys* 51, 1 (2018), 4:1–4:27.

[4] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. 2003. Fault-based testing without the need of oracles. *Information and Software Technology* 45, 1 (2003), 1–9.

[5] Darryl C. Jarman, Zhi Quan Zhou, and Tsong Yueh Chen. 2017. Metamorphic Testing for Adobe Data Analytics Software. In *Proceedings of the IEEE/ACM 2nd International Workshop on Metamorphic Testing (ICSE MET '17), in conjunction with the 39th International Conference on Software Engineering (ICSE '17).* 21–27. https://doi.org/10.1109/MET.2017.1

[6] William M. McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/toc.htm

[7] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824.