Performance Metamorphic Testing: Motivation and Challenges

Sergio Segura, Javier Troya, Amador Durán and Antonio Ruiz-Cortés
Department of Computer Languages and Systems
Universidad de Sevilla, Seville, Spain
{sergiosegura,jtroya,amador,aruiz}@us.es

***Abstract*—Performance testing is a challenging task mainly due to the lack of *test oracles*, that is, mechanisms to decide whether the performance of a program under a certain workload is either acceptable or poor due to a performance bug. Metamorphic testing enables the generation of test cases in the absence of an oracle by exploiting the relations (so–called *metamorphic relations*) between the inputs and outputs of multiple executions of the program under test. In the last two decades, metamorphic testing has been successfully used to detect functional faults in a variety of domains, ranging from web services to simulators. However, the applicability of metamorphic testing to detect performance bugs is a topic that remains unexplored. In this vision paper, we introduce *Performance Metamorphic Relations (PMRs)* as expected relations between the performance measurements of multiple executions of the program under test. We hypothesize that these relations can be turned into assertions for the automated detection of performance bugs removing the need for complex benchmarks and domain experts guidance. As a further benefit, PMRs can be turned into fitness functions to guide search-based techniques on the generation of test data that violate the relations, revealing bugs. This novel idea is motivated with examples and an overview of some of the challenges in this promising topic.**

***Keywords*-Metamorphic testing, performance testing, search-based testing**

I. Introduction

Performance testing aims to reveal programming errors that cause significant performance degradation in the system under test [1], e.g., excessive memory consumption. Performance defects are very common in released software programs. For example, Mozilla developers fix between 5 and 60 performance bugs reported by users every month [2]. Similarly, the emerging trend of mobile applications brings new challenges in terms of performance testing like detecting the infamous energy leaks or memory bloats [3], [4]. Overall, performance bugs cause poor usability and a waste of resources, which might lead to loss of users or hundred-million-dollar software projects to be abandoned [1], [2].

In contrast to functional bugs, performance bugs do not produce wrong results or crashes in the program under test and therefore they cannot be detected by simply inspecting the program output. For example, suppose that a browser takes 200ms to render a given Web page: Is this the expected performance? How slow should it be considered as a performance bug? Answering these questions requires not only a good knowledge of the application, but also considering other aspects as the computer hardware or its current workload. Compared to functional faults, performance bugs are significantly harder to detect and require more time and effort to be fixed [1]. This is partly due to the lack of *test oracles*, that is, mechanisms to decide whether the performance of the program under a certain workload is acceptable i.e., the *oracle problem*. Typical oracles in performance testing are human judgement, often involving long discussion among developers, or comparisons among different programs with similar functionality (or different versions of the same program) [1]–[3].

The lack of automated oracles is recognized as one of the key challenges on performance testing. Jin et al. [2] conducted an empirical study of 109 real-world performance bugs and concluded that "*techniques that can smartly compare performance numbers across inputs and automatically discover the existence of performance problems are desired*". Liu et al. [3] studied 70 real-world performance bugs in Android applications and reached the same conclusion: "*effective performance testing needs automated oracles to judge performance degradation*". Similarly, Nistor et al. [1] analysed 210 performance bugs from three mature open source projects and concluded that "*better oracles are needed for discovering performance bugs*".

Metamorphic testing alleviates the oracle problem by providing an alternative when the expected output of a test execution is unknown [5]. Rather than checking the output of an individual program execution, metamorphic testing checks whether multiple executions of the program under test fulfil certain necessary properties called *metamorphic relations*. For instance, consider the program $merge(L_1, L_2)$ that merges two lists into a single ordered list. The order of the parameters should not influence the result, which can be expressed as the following metamorphic relation: $merge(L_1, L_2) = merge(L_2, L_1)$. A metamorphic relation comprises of a so-called *source test case* (L_1, L_2) and one or more *follow-up test cases* (L_2, L_1), derived from the source test case. A metamorphic relation can be instantiated into one or more *metamorphic tests* by using specific input values, e.g., $merge([2, 3], [1, 5]) = merge([1, 5], [2, 3])$. If the outputs of a source test case and its follow–up test case(s) violate the metamorphic relation, the metamorphic test is said to have failed, indicating that the program under test contains a bug. In a recent survey, Segura et al. [6] reviewed about 120 papers on metamorphic testing and identified successful

applications of the technique in a variety of domains, ranging from web services to compilers. Interestingly, it was found that all the reviewed papers focused on the use of metamorphic testing for the detection of functional faults. Therefore, the potential application of metamorphic testing for the detection of performance bugs remains unexplored.

In this paper, we present the following hypothesis:

> *Metamorphic testing is a helpful technique to address the oracle problem in performance testing, supporting the automated detection of performance bugs.*

The intuitive idea is as follows. Let us suppose the call $merge(l_1, l_2)$ takes 300ms to provide an output, with l_1 and l_2 being two specific lists: Is this correct? Hard to say. Intuitively, the execution time required to merge the lists should be equal or greater if more elements are added to both lists. This can be expressed as the following *Performance Metamorphic Relation (PMR)*: $T(merge(L_1, L_2)) \leq T(merge(L_1 \cup L_3, L_2 \cup L_4))$, where L_3 and L_4 are two lists containing k random items each, with $k > 0$. Based on this, the following metamorphic test can be constructed: $T(merge(l_1, l_2)) \leq T(merge(l_1 \cup l_3, l_2 \cup l_4))$, with l_3 and l_4 being two lists with 1000 random numbers each. A key benefit of PMRs is that they are independent of the inputs selected, i.e., the relation should be satisfied for any L_{1-4}. Also, if the source and follow-up test cases are executed in the same computer and one immediately after the other, they should be equally affected by external factors such as the hardware settings or the computer workload, and the relation should still hold (note that issues like cache warmup should still be handled). Thus, PMRs may be turned into assertions for the automated detection of performance bugs, removing the need for complex benchmarks and human judgement.

As a further benefit of the proposal, PMRs can be turned into fitness functions to guide search-based techniques on the generation of test data that violate the relations. For instance, the previous PMR could be translated into the following fitness function (to be maximize): $(T(merge(L_1, L_2)) - T(merge(L_1 \cup L_3, L_2 \cup L_4)))$. Intuitively, this function could be used to search for inputs where the execution time of the source test case is greater than the execution time of the follow-up test case. In other words, the search could be guided toward inputs where the relation is violated as much as possible. In practice, this means that the approach would not only alleviate the oracle problem in performance testing, but it would also enable the use of smart techniques for the automated detection of performance bugs.

In what follows we first present some motivating examples inspired by real-world performance bugs in Section II. Section III introduces some of the challenges related to the definition of PMRs. The management of false positives and false negatives also raises open questions described in Sections IV and V respectively. Section VI discusses the problem of test data generation and how PMRs can be turned into fitness functions to guide search-based testing techniques. Finally, Section VII summarizes our main conclusions.

II. Motivating examples

In this section we present several PMRs inspired by real performance bugs. Each relation is intentionally presented in a simple and naive way for the sake of understandability, and some of them are studied deeper in the next sections.

BookmarkAll. Users reported that Firefox took too long when they clicked the "bookmark all (tabs)" with a large number of open tabs[1] [2]. The problem was caused by the use of separate database transactions for bookmarking each tab, which progressively degraded performance as the number of tabs increased. The bug was fixed by batching all the tasks into a single transaction. Inspired by this bug, the following PMR could be defined:

$$T(bookmarkAll(x)) \leq T(bookmarkAll(y)) \qquad (\textbf{\textit{PMR}}_1)$$

where x and y are positive integers representing the number of tabs to bookmark, and $x < y$. Intuitively, the relation expresses that the execution time of the operation is expected to increase with the number of open tabs to be bookmarked.

LoadImg. Several Chrome users reported memory leaks when manipulating images, which was classified as a bug[2]. This bug caused unexpected levels of memory usage when loading images of different sizes. Rendering large images was expected to consume more memory than rendering small images. However, if a small image was loaded after a bigger one, the memory usage increased. This was due to problems with the garbage collector, which did not work when it should. Inspired by this bug, the following PMR could be defined:

$$M(loadImg(img_1)) \geq M(loadImg(img_2)) \qquad (\textbf{\textit{PMR}}_2)$$

where img_2 is an image derived from img_1 but with a smaller size, for instance cropping it or decreasing its quality. This relation expresses that loading an image with a specific size should consume more memory, or the same, than when a smaller image is loaded.

UpdateGUI. Zmanim is a location-aware application for reminding Jewish people about prayer time during the day. The app sends alerts according to users' locations and corresponding time zones. Users noted an excessive consumption of battery power (i.e., energy leak) caused by a defect in the code[3] [3]. In certain circumstances the application kept receiving location changes to update its GUI even when the application was switched to background, wasting valuable battery power. Inspired by this performance bug, the following PMR could be defined:

$$E(updateGUI(t, \text{`}active\text{'})) > E(updateGUI(t, \text{`}paused\text{'})) \quad (\textbf{\textit{PMR}}_3)$$

where *t* is a certain usage test pattern (e.g., sequence of user actions) and the second input value indicates the final state of the app, either 'active' or 'paused'. Intuitively, the relation indicates that the mobile application should consume more energy when it is active than when it is paused.

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=490742
[2] https://bugs.chromium.org/p/chromium/issues/detail?id=337425
[3] https://zmanim.myjetbrains.com/youtrack/issue/Z-50

III. Defining performance metamorphic relations

The rationale behind metamorphic testing is that bugs can be exhibited when observing the differences among two or more program executions with different input values, e.g., $merge(L_1, L_2) = merge(L_2, L_1)$. However, it is unclear to what extent performance bugs can be exposed with certain input values and remain undetected with others.

To explore this issue we reviewed some of the recent literature on performance testing. Jin et al. found out that two thirds of the performance bugs need inputs with special features to manifest [2]. For instance, to trigger the bug in Firefox, the user has to click "bookmark all" with many open tabs. Analogously, Liu et al. [3] discovered that one third of the bugs required special user interactions in order to be revealed. For instance, the Zmanim's energy leak needs the following steps to be reproduced [3]: (1) switch on GPS, (2) configure Zmanim to use current location, (3) start its main activity, and (4) hit the "Home" button when GPS is acquiring a location. These findings suggest that a significant portion of performance bugs are revealed when exercising the program with certain inputs only, which means that some defect could be exhibited when comparing the behaviour of source and follow-up test cases. Interestingly, this is in line with the conclusions of Jin et al. [2], who wrote: "*These bugs [performance bugs] cannot be effectively exposed if software testing executes each buggy code unit only once, which unfortunately is the goal of most functional testing*".

However, some other results suggest that the applicability of performance metamorphic testing might be limited. Jin et al. [2] reported that a portion of bugs (15 out of 109 in their study) are always active because they are located in code exercised by all inputs, e.g., start-up or shutdown phases. In practice, this means that PMRs would be rarely able to detect performance bugs of this type, since the fault would affect equally the source and follow-up test cases. For instance, suppose a fault in the start-up phase of Firefox causing database transactions to take ten times longer than expected. This bug would equally affect the source and follow-up test cases in PMR_1, and thus the relation would be satisfied with any input, remaining the bug undetected. This yields to the following challenges:

> *Challenge 1: Identify PMRs in different application scenarios, preferably on realistic settings, and show their effectiveness at detecting performance defects.*

> *Challenge 2: Develop guidelines for the definition of PMRs. These could be domain-dependent and hopefully backed by studies on the characteristics of the typical performance bugs reported on each domain.*

IV. Managing false positives

In functional metamorphic testing, most metamorphic relations are defined for *deterministic* programs where, for certain inputs, the relation is either satisfied or violated, e.g., $merge([2,3],[1,5]) = merge([1,5],[2,3])$. In contrast, the measurement of non-functional properties such as execution time, memory consumption or energy usage is inherently *non-deterministic*. For instance, the battery power consumed by a mobile application could vary from one execution to another due to the device workload, communication issues or automated updates. In practice, this means that PMRs could be sometimes violated without that being an indicator of a performance bug, what results in a *false positive*.

A few approaches have addressed the problem of testing non-deterministic programs using metamorphic testing. Guderlei and Mayer [7] proposed *Statistical Metamorphic Testing*. The method works by generating two or more sequences of outputs by executing source and follow–up test cases. Then, the sequences of outputs are compared according to their statistical properties using statistical hypothesis tests. Murphy et al. [8] argued that in certain cases slight variations in the outputs are not actually indicative of errors, e.g., floating point calculations. To address this issue, the authors propose the concept of *heuristic test oracles*, by defining a function that determines whether the outputs are "close enough" to be considered equal. Inspired by these approaches, we envision at least two complementary methods to manage false positives caused by non-determinism on performance metamorphic testing, namely:

1) Violation threshold: This approach consists in running each metamorphic test multiple times and setting a threshold in the number of violations that we consider admissible. For instance, let us suppose that we run a metamorphic test 100 times with a violation threshold of 5%. If the number of violations is greater than 5 the PMR is violated, otherwise it is satisfied.

2) Relation thresholds: This approach, inspired by [8], consists in setting thresholds to allow certain differences in the performance measurements of source and follow-up test cases. PMR'_1 depicts a refined version of PMR_1 using this approach, where β represents the threshold for the comparison. The value of β could be set to an absolute value (e.g., 100ms) or a relative time value with respect to the source and follow-up test cases, e.g., $y \times T(bookmarkAll(x))/x$. As an example, suppose we run a source test case with $x = 5$, observing an execution time of 250ms. Next, a follow-up test case is run with $y = 7$ and an execution time of 210ms. This small difference (40ms) may not indicate a bug, since external factors may influence the execution time. However, it violates PMR_1, producing a false positive. Setting $\beta = 350ms$, however, the relation PMR'_1 would be satisfied, being the observed difference not significant to be considered a failure, i.e., $250 - 210 \leq 350$.

$$T(bookmarkAll(x)) - T(bookmarkAll(y)) \leq \beta \quad (PMR'_1)$$

All this can be summarized in the following challenges:

> *Challenge 3: Evaluate the feasibility of using violation and relation thresholds to avoid false positives and provide guidelines for the definition of effective threshold values.*

Challenge 4: Propose new methods to address false positives on the definition of PMRs.

V. Managing false negatives

Analogously to the false positives produced by non–determinism, PMRs could also produce *false negatives*, that is, situations where the relation is satisfied despite the program being faulty. Consider the *BookmarkAll* bug as an example. According to the bug report, the time required for the browser to execute the task increased dramatically with the number of tabs to be bookmarked. This could lead to the following situation: suppose that Firefox takes 250ms to bookmark 5 tabs, and 15000ms to bookmark 20 tabs. The relation $T(bookmarkAll(5)) \leq T(bookmarkAll(20))$ would be satisfied, despite the large difference in the execution times ($250 \leq 15000$), remaining the fault undetected.

As illustrated in the previous section with false positives, a sensible approach to address false negatives could also be the use of thresholds. PMR_1'' depicts a new version of the relation where α represents a threshold for the comparison. Again, the value of α could be set to an absolute or a relative time value with respect to the source and follow-up test cases, for instance $(x - y) \times 500$. Continuing with the previous example, setting $\alpha = -7500$ and $\beta = 350$, the relation would be violated ($-7500 \leq (250 - 15000) \leq 350$), suggesting the presence of a performance bug.

$$\alpha \leq T(bookmarkAll(x)) - T(bookmarkAll(y)) \leq \beta \quad (PMR_1'')$$

From this, the following challenges are identified:

Challenge 5: Evaluate the feasibility of using thresholds to identify false negatives and provide guidelines for the definition of effective threshold values.

Challenge 6: Propose new methods to address false negatives on the definition of PMRs.

VI. Test data generation

Detecting performance bugs by means of testing requires finding test inputs that manifest the unexpected performance behaviour in the program under test. This can be extremely challenging [1]–[4]. Consider, for example, the specific sequence of user interactions needed to trigger the Zmanim's energy leak, described in Section III. Although this work does not address the problem of test data generation in performance testing, we envision that PMRs could help on the search of effective test inputs. This is because unlike functional metamorphic relations, where the outcome is Boolean (either satisfied or violated), PMRs can be translated to a numeric result that reflects to what extend the relation is satisfied or violated. In practice, this means that PMRs can be turned into fitness functions to be used in search-based testing techniques. For example, PMR_2 can be turned into the following fitness function (to be minimized):

$$M(loadImg(img_1)) - M(loadImg(img_2))$$

In practice, this fitness function would favour those images where the memory consumed by the source test cases is less than the memory consumed by the follow-up test cases. In other words, the function would guide the search toward input images that violate the PMR to the maximum possible extent, revealing potential defects. This leads to the following challenge:

Challenge 7: Propose and evaluate search-based methods for the automated generation of test data for performance metamorphic testing.

VII. Conclusions

The automated detection of performance bugs is recognized as a relevant and challenging problem. In order to address such problem, this vision paper opens the path to a promising research topic that has remained unexplored so far, performance metamorphic testing. We have sketched how it could effectively reveal performance bugs reported by users in well-known real-world applications, and have enumerated some of the challenges that need to be resolved for the successful application of performance metamorphic testing.

Acknowledgment

This work has been supported by the Spanish Government under CICYT project BELI (TIN2015-70560-R), the Excellence Network SEBASENet (TIN2015-71841-RED), and the Andalusian Government project COPAS (P12-TIC-1867).

References

[1] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," in *Proc. of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, pp. 237–246.

[2] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and Detecting Real-world Performance Bugs," in *Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 77–88.

[3] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 1013–1024.

[4] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proc. of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 588–598.

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

[6] S. Segura, G. Fraser, A. Sanchez, and A. Ruiz-Cortes, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[7] R. Guderlei and J. Mayer, "Statistical metamorphic testing: Testing programs with random output by means of statistical hypothesis tests and metamorphic testing," in *Proc. of 7th International Conference on Quality Software, 2007. QSIC '07*, 2007, pp. 404–409.

[8] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," in *Proc. of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. ACM, 2009, pp. 189–200.