Information and Software Technology 98 (2018) 1–4

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Performance metamorphic testing: A Proof of concept

Sergio Segura*, Javier Troya, Amador Durán, Antonio Ruiz-Cortés

Departamento de Lenguajes y Sistemas, Informáticos Universidad de Sevilla, Spain

ARTICLE INFO

Keywords:
Metamorphic testing
Performance testing
Search–based testing

ABSTRACT

Context: Performance testing is a challenging task mainly due to the lack of *test oracles*, i.e. mechanisms to decide whether the performance of a program is acceptable or not because of a bug. Metamorphic testing enables the generation of test cases in the absence of an oracle by exploiting the 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 different domains. However, its applicability to performance testing remains unexplored.

Objective: We propose the application of metamorphic testing to reveal performance failures.

Method: We define *Performance Metamorphic Relations (PMRs)* as expected relations between performance measurements of multiple executions of the program under test. 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.

Results: The feasibility of the approach is illustrated through an experimental proof of concept in the context of the automated analysis of feature models.

Conclusion: The results confirm the potential of metamorphic testing, in combination with search-based techniques, to automate the detection of performance bugs.

1. Introduction

Performance testing [1] aims to reveal errors that cause significant performance degradation in the program under test (PuT). Performance defects are very common in released software programs. For example, Mozilla developers fix between 5 and 60 user–reported performance bugs every month [2]. Similarly, mobile applications bring new challenges like detecting energy leaks or memory bloats [3,4].

In contrast to functional bugs, performance bugs do not produce wrong results or crashes in the PuT and therefore cannot be detected by simply inspecting the program output. Therefore, they are significantly harder to detect and require more time and effort to be fixed [1]. This is mainly due to the lack of *test oracles*, i.e. mechanisms to decide whether the performance of a program under a certain workload is acceptable or not. Typical oracles in performance testing are human judgement or comparisons among different programs with similar functionality [1–3], which are far from trivial.

Metamorphic testing alleviates the oracle problem by checking whether multiple executions of the PuT fulfil certain necessary properties called *metamorphic relations*. For instance, consider the program *merge* (L_1, L_2) that merges two ordered lists into a single ordered list. The parameter order 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 one *source test case* (L_1, L_2) and one or more *follow–up test cases* (L_2, L_1). Each metamorphic relation can be instantiated into one or more *metamorphic tests* by using specific inputs, e.g. $merge([2, 3], [1, 5]) = merge([1, 5], [2, 3])$. If the outputs of the source test cases and the follow–up test cases violate the relation (*equality* in this example), the test is said to have failed, indicating that the PuT contains a bug.

Recent surveys have reviewed the large body of papers on metamorphic testing and identified successful applications of the technique in a variety of domains, ranging from web services to compilers [5,6]. Interestingly, however, it has been found that all the reviewed papers focused on the detection of functional faults, with remarkable applications to areas such as proving, validation and quality assessment. Therefore, the potential application of metamorphic testing for the detection of performance bugs remains unexplored.

In a previous paper [7], we proposed the application of metamorphic testing to reveal performance failures, and we presented some of the many challenges related to it. In this short paper, we go a step further by confirming the feasibility of the approach in a realistic

* Corresponding author.
E-mail address: sergiosegura@us.es (S. Segura).

https://doi.org/10.1016/j.infsof.2018.01.013
Received 4 December 2017; Received in revised form 22 January 2018; Accepted 29 January 2018
Available online 31 January 2018
0950-5849/ © 2018 Elsevier B.V. All rights reserved.

scenario.

2. Performance metamorphic testing

Let us suppose that $merge(l_1, l_2)$ takes 300 ms 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 T represents the execution time, and L_3 and L_4 are two nonempty lists containing k random items. Based on this, metamorphic tests such as $T(merge(l_1, l_2)) \leq T(merge(l_1 \cup l_3, l_2 \cup l_4))$ could be applied. A key benefit of PMRs is their independence of the selected inputs, i.e. the previous one should be satisfied for any list. Thus, PMRs may be turned into assertions for the automated detection of performance bugs, removing the need for complex benchmarks and human judgement.

Real performance bugs can also inspire PMRs [7]. For example, some users of the Chrome browser reported unexpected levels of memory usage when loading images of different sizes.[1] Rendering large images was expected to consume more memory than rendering small images. However—due to problems with the garbage collector—if a small image was loaded after a bigger one, the memory usage increased. Inspired by this bug, the following PMR could be defined:

$$M(loadImg(img_1)) \geq M(loadImg(img_2)) \qquad (PMR_1)$$

where M represents the memory consumed, and img_2 is an image derived from img_1 but with a smaller size, for instance cropping it or decreasing its quality.

2.1. 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 inputs. However, it is unclear to what extent performance bugs can be exposed with certain input values and remain undetected with others.

Recent works have drawn conclusions that make us foresee the usefulness of applying metamorphic testing in this context. In particular, Jin et al. found out that two thirds of the performance bugs need inputs with special features to manifest [2], and Liu et al. [3] discovered that one third of the bugs required special user interactions in order to be revealed. These findings suggest that a significant portion of performance bugs are revealed when exercising the program with certain inputs only.

2.2. Managing false positives and false negatives

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*. Analogously, PMRs could also produce *false negatives*, i.e. situations where the relation is satisfied despite the PuT being faulty.

In our previous work, we discussed different alternatives to address false positives and false negatives, including *tolerance thresholds* to allow certain differences in the performance measurements of source and follow–up test cases [7]. For example, considering PMR_1, false positives could be mitigated by defining the following PMR using a threshold β:

$$M(loadImg(img_2)) - M(loadImg(img_1)) \leq \beta$$

which means that the relation will only be marked as violated when the memory consumed by img_2 is greater than the memory consumed by img_1 by an amount of β or larger. The value of β could be set to an absolute value (e.g. 100KB) or a relative value (e.g. 10%).

2.3. Test data generation

Detecting performance bugs by means of testing requires finding test inputs that manifest the unexpected performance behavior in the program under test, what can be extremely challenging [1–4]. 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 instance, PMR_1 can be turned into the following fitness function (to be maximized):

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

This fitness function would guide the search towards input images where the memory consumed by the source test case (large image) is lower than the memory consumed by the follow-up test case (small image), i.e. images that violate the PMR to the maximum possible extent, revealing potential defects.

3. Proof of concept

In this section, we present a proof of concept by studying the feasibility of the approach in a realistic scenario.

3.1. Subject program

We used SPLAR [8], a popular tool for the automated analysis of feature models, the *de-facto* standard for variability modelling in software product lines. A *feature model* is a tree-like structure that represents software products in terms of features (nodes) and constraints among those features (edges) [9]. SPLAR takes a feature model as input, translates it into a Boolean formula represented by a Binary Decision Diagram (BDD), and uses an off-the-shelf BDD solver to extract information from the model, e.g. check model consistency.

3.2. Seeded fault

A key property of BDDs is that they provide fast analysis times, but at the cost of memory usage and preprocessing time. SPLAR provides two key parameters to control how the BDD is built, namely the initial size of the table to store BDD nodes and the cache size, both set to 10K by default. The size of the actual BDD strongly depends on the size of the input feature model. Setting too high or too low values for these

Table 1
Execution times (ms) with random feature models.

Features	Min	Avg	Max
100	0	1	21
150	0	4	2690
200	0	30	22,463

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=337425.

Download English Version:

https://daneshyari.com/en/article/6948050

Download Persian Version:

https://daneshyari.com/article/6948050

Daneshyari.com