Who Guards the Guards?
On the Validation of Test Case Migration

Ivan Jovanovikj, Enes Yigitbas, Anthony Anjorin, Stefan Sauer
Paderborn University, Germany
Zukunftsmeile 1, 33102 Paderborn
{ivan.jovanovikj|enes.yigitbas|anthony.anjorin|sauer}@upb.de

Abstract

Software migration, as a well-established strategy to reuse software, results in a software system that runs in a new environment but exhibits the same behavior as before the migration. To ensure behavioral preservation, existing test cases can be used to safeguard the software migration. This implies two things: test cases have to be co-migrated with the system and, after the migration, they have to be validated as well. Similarly as for system migration, behavioral preservation is a must for test case migration, i.e., the migrated test cases still have to assert the same expected system behavior as the old test cases. Despite the importance of validating test case migration, the area is not yet well researched. In this paper, we analyze the challenges in validating test case migration and propose mutation analysis as a suitable validation technique.

1 Introduction

Software migration is a well-established approach for transferring old software system to a new environment. A crucial requirement to be fulfilled by the migrated system is the behavioral equivalence with the old system. Asserting a system's behavior can be done by applying software testing. As testing is costly and time-consuming, reusing test cases, whenever is possible, is highly desired. Reusing test cases means co-migrating them together with the system. This means transferring them to a new environment without changing their "functionality", i.e., without changing the expected behavior the test cases assert.

As such migrated test cases are used as safeguards for the system migration, their correct migration is crucial. But, test case migration is far from trivial as several challenges have to be addressed [1]. Additionally, the test case migration is tightly coupled with the system migration which makes its validation especially challenging. This area, to the best of our knowledge, is currently not yet sufficiently researched. It is still unclear which techniques can be used, and what their potential and limitations are. The following question is thus still open for investigation: What is the safeguard for test case migration?

Validating test case migration is similar to validating test case refactoring as both activities tend to keep asserted behavior unchanged. Mutation analysis has been applied to safeguard test refactoring and this actually inspired us to apply this technique to validate test case migration as well.

In this paper, we discuss the challenges involved in applying mutation analysis as a validation technique in this context, sketch our solution idea, and explain how it addresses the identified challenges.

2 Validation of Test Case Migration and Involved Challenges

Refactoring is defined as the process of improving the internal structure of a system without changing its observable behavior. Similarly, test case refactoring can be seen as improving the internal structure of a test case without changing its "observable behavior", i.e., without changing the behavior that it asserts. Regarding system refactoring, a set of test cases with adequate coverage can be used as a safeguard. Ensuring the correctness of test case refactoring is, however, a bit more challenging. A refactored test case still has to pass after the refactoring as it is being executed against a correct system (assuming the test cases passed before refactoring). However, one should additionally ensure that it still properly detects incorrect system behavior. This is known as a problem of false negatives and false positives. A test case should only fail when a problem exists (true positive) and pass when a certain problem does not exist (true negative). Mutation analysis has been seen as a solution to this problem, as it measures the capability of test cases to properly detect unwanted system behavior. Mutation analysis is a technique used for the creation of new test cases as well as for quality evaluation of existing test cases [2]. It involves the modification (mutation) of an existing system by making small syntactic changes to create mutants. In other words, small faults are seeded in the system and then existing test cases are executed against the mutants. In general, there are two possible outcomes: (i) at least one test case "detects" the change introduced in the mutant, i.e., the mutant was killed or (ii) no test case "detects" the change, i.e, the mutant survived. A mutant survived because: (i) it is an equivalent mutant, or (ii) the test cases were not able to detect the seeded fault. A mutation score (the ratio of the total number of killed mutants to the total number of non-equivalent mutants) indicates the quality of the test cases.Applying mutation analysis in test migration involves additional challenges, however, due to the fundamental difference between refactoring and migrating test cases; When refactoring tests the system does not change, while when co-migrating tests *both* the

tests and the system are changed. In the following, we identify these challenges:

(C1) Compared to refactoring, where the refactored test cases are executed against the same system, the migrated test cases after are executed against the migrated system. There is no guarantee that the system's behavior was correctly preserved. **(C2)** The test cases must be changed as part of the migration, as they have to be adapted to the migrated system and the new testing environment. There is no guarantee that their behavior is correctly preserved. **(C3)** The object under mutation can be chosen from a wide range of artifacts or activities depending on the concrete context. The decision which what should be mutated is important as it can influence the effectiveness and efficiency of the complete validation method. **(C4)** Once applied, mutation analysis results in a mutation score which should be an indicator for the correctness of the test case migration. This score has to be interpreted carefully to show the eventual weak points of the test case migration.

3 Solution Idea

Inspired by the application of mutation analysis in test case refactoring, we performed an initial analysis on applying the same solution to test case migration. As shown in Figure 1, we have analyzed a simple constellation of test case and system migration which are performed together. Depending on what is being mutated, we have identified 6 different scenarios in total. Each scenario has an assumption part (e.g., that a certain mutation framework exists) and an indication part, where the results of applying mutation analysis are discussed. From a mutation analysis perspective, typical scenarios are when the system under test is mutated. As both the migrated and old system can be mutated in our case, we have two such scenarios, namely *Scenario 1* and Scenario 2 (Figure 1). In *Scenario 3*, the system migration is mutated which can be considered as an indirect mutation of the migrated system. Furthermore, mutation analysis can also be used to mutate test cases. In *Scenario 4* and *Scenario 5*, migrated test cases and old test cases are mutated. The last scenario, *Scenario 6*, is the mutation of the test case migration, a variant of *Scenario 4*, as it is an indirect mutation of the migrated test cases.

The challenges **C1**, **C2**, and **C3** are addressed by the different scenarios which we have introduced. Each scenario is suitable for a specific context which is matched by the assumption part. Analogously to the general case of mutation analysis [2], we assume that the programmers (in this case the migration specialists) are competent and that they tend to implement migration transformations that are already close to the correct migration transformations. This assumption is important when addressing **C1** and **C2**. Regarding **C3**, one can consider which mutation scenario is the best by, for example, comparing the avail-

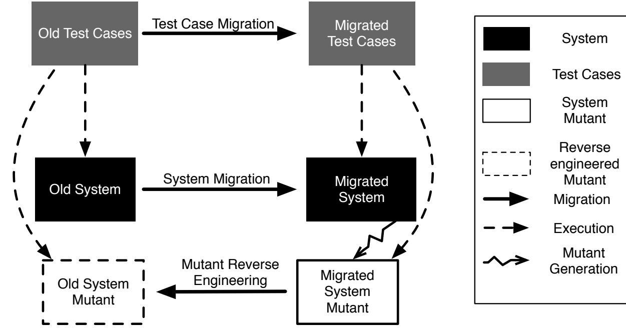

Figure 1: Mutation of Migrated System.

able mutation frameworks. In the following, we focus on *Scenario 2*, shown in Figure 1, in which the migrated system is being mutated. We assume that a suitable mutation framework for the migrated system exists. Additionally, we assume that it is possible to derive old system mutants from migrated system mutants via reverse engineering. The migrated test cases are executed against the mutant of the migrated system. There are two possible outcomes for each mutant: a mutant is either killed or not. In the following we briefly show how challenge **C4** is addressed by discussing the mutation score. We firstly analyze the case when the migrated system mutant is killed. If a corresponding old system mutant, obtained via reverse engineering, is killed as well, then this represents the expected case that increases trust in the test case migration. If the obtained old system mutant was not killed, then it is either equivalent or non-equivalent. If it is equivalent, then there are no indications. If it is non-equivalent, then it means that we have an erroneous old system not detected by the old test cases. Revisiting *Scenario 1* could help fixing the old test suite, by augmenting them with new test cases. The other half of the analysis deals with the case when the migrated system mutant is not killed. If the reverse engineered old system mutant is killed, it suggests that at least one migrated test case is a false negative. If the reverse engineered old system mutant is not killed and it is equivalent, than no indications can be derived. If the reverse engineered old system mutant is not equivalent, then the quality of the old test cases has to be checked.

Our results show that mutation analysis can provide useful information, i.e., indications about eventual problems in test case and system migration.

References

[1] I. Jovanovikj, M. Grieger, and E. Yigitbas. Towards a model-driven method for reusing test cases in software migration projects. *Softwaretechnik-Trends*, 2016.

[2] R. J. Lipton and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 1978.