Towards a Model-Driven Method for Reusing Test Cases in Software Migration Projects

Ivan Jovanovikj, Marvin Grieger, Enes Yigitbas
s-lab – Software Quality Lab, Paderborn University
Zukunftsmeile 1, 33102 Paderborn
{ijovanovikj, mgrieger, eyigitbas}@s-lab.upb.de

1 Introduction

Software testing is a very important activity in software migration as it verifies whether the migrated system still provides the same functionality as the legacy system. However, testing in a software migration project is a costly and time-consuming endeavour [1]. Therefore, when an existing set of test cases is available, its reuse should be considered. This can be beneficial, not just from economical perspective, but also from practical perspective: the existing test cases contain valuable information about the functionality of the legacy system and therefore about the desired functionality of the migrated system, too.

However, reusing existing test cases is far from trivial since several challenges need to be addressed. For example, the *quality* of the existing test cases needs to be assessed, since they might have become legacy, too. If test cases are redundant or do cover parts of the system that are not used anymore, there is no value in spending effort on reusing them. As another example, the *traceability* between the test cases and the legacy system needs to be established. Only then, changes to the legacy system can be reflected to the test cases.

To address these challenges, we envision a novel test case migration method that is based on test case re-engineering: Following the idea of model-driven software migration, we extract models out of the existing test cases, reflect the changes from the software migration (co-evolution), and at the end, we employ model-based testing to generate test cases for the migrated system. Such a test case re-engineering method that enables co-evolution of test cases, is not fully covered by the existing research.

In this paper, we first discuss a set of challenges that a test case re-engineering method needs to address. Thereafter, we sketch our method and describe in which way it considers the identified challenges.

2 Challenges in Test Case Reuse

After observing test case reuse in practice and analyzing current research, publications as well as migration projects, we identified a set of challenges which should be addressed by an approach that aims to provide maximal reuse of the existing test cases.

C1. Assessment of existing test cases. The value of the existing test cases has to be initially assessed. This assessment is important to see whether further reuse is beneficial. It should include checking quality characteristics like effectiveness, understandability, or structuredness, identifying the relation to the system or code components, evaluating the test coverage, etc.

C2. Consideration of all test case levels. Test cases at all test levels (unit, integration, system level) have to be considered. Depending on the type of the migration (language, framework or architectural change), test cases from all test levels may be needed in order to test the migrated system.

C3. Consideration of automated and non-automated test cases. Both automated and non-automated test cases have to be considered. A non-automated test case may still contain valuable information about the migrated system. For example, system tests are mostly manual, since they address the expected behavior from the end users perspective.

C4. Support for different levels of granularity. In model-based testing, one of the key issues is the level of granularity of test models. Should the test cases be more coarse grained and contain only the important concerns (higher level of abstraction) or should they be more fine-grained, each covering more specific concern in detail (lower level of abstraction).

C5. Establishment of traceability. A relation between the test models and the models of the system has to be established, as shown in Fig. 3. This relation enables an assessment of test coverage and also, propagation of relevant system changes. For example, using these relations, changes in the system architecture could be reflected to the integration test model.

C6. Refactoring of test models. Refactoring of the obtained test models my be needed since the existing test case set may contain obsolete or duplicate test cases. We assume that these anomalies could be easier detected at model level, as shown in Fig. 2. For example, an integration test case may check a connection of two components that actually should not be connected, or in system testing, a test case may check a scenario that should not be possible.

C7. Reflection of system changes. The changes happening during the restructuring of the software migration have to be reflected to the test models. It is important in order to obtain relevant test models, and thus, relevant test cases. In other

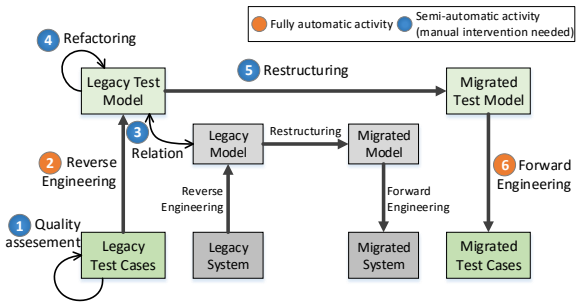

Figure 1: Co-evolution of test cases by applying a test case re-engineering method

words, the co-evolution of test cases needs to be supported. Establishing traceability, as explained under C5, is an essential prerequisite to enable co-evolution of test cases.

3 Solution Idea

Having the challenges in mind, we present our envisioned solution and discuss how we aim to address these challenges.

In general, we envision a test case migration approach, i.e., a test case re-engineering method, tightly related to the software migration as shown in Figure 1. As the first step, we do an initial *assessment* of the existing test set regarding its quality (C1), to decide whether reuse of existing test cases would be beneficial. For example, by analyzing previous test reports, we can find out how effective these test cases were and what coverage they had, regarding requirements, code components, etc.

If the assessment step gives a positive result, we can then approach to the second step, namely the *reverse engineering* of test cases. Existent reverse engineering techniques from model-based testing could be re-used [2], or, in case of specific requirements (different test levels, non-automated test cases, different levels of granularity), a new technique may be developed. For the non-automated test cases, on a system level for example (cf. Fig. 2), we may need a technique which would also be able to discover models at different level of granularity (C2, C3 and C4). The outcome of the *reverse engineering* step at the level of unit testing could be a class, activity or state-chart diagram, at the level of integration testing it could be component diagram (cf. Fig. 3), while activity or state chart diagrams could be used at the level of system testing (cf. Fig. 2).

After the test models are obtained, as the third step, we establish *relations* between test models and models of the system (C5). We assume that by using model matching techniques, corresponding model elements can be be related, as shown in Fig. 3. This should enable *refactoring* of test models when some errors are detected, as well as *reflection* of the changes that happen in the system.

Once the *relation* is established, in the fourth step we analyze the test models again to see whether *refactoring* is needed (C6). For example, for the system level test cases, after the re-engineering step we obtain an activity diagram representing the expected behavior according to the old test cases. Due to evolutionary development of the old test cases, it may be the case that some non-allowed paths are present in the diagram, which could be easily noticed and corrected. For example, in Fig. 2, the path including *activity D* should not be checked anymore, since this activity is not part of the desired flow anymore.

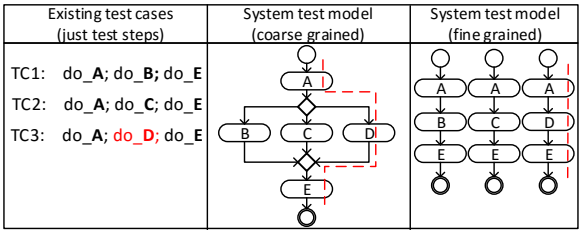

Figure 2: Example for error detection and level of granularity

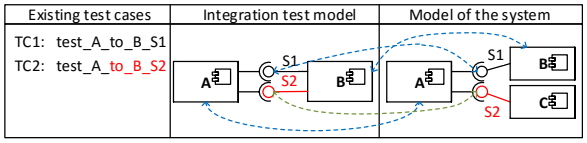

Figure 3: Example for relations between an integration test model and a model of the system

In the fifth step, we do *restructuring* of the test models by reflecting relevant changes that happen during restructuring of the system (C7). In different migration scenarios, different levels of test cases are affected in different ways. For example, Fig. 3 gives an example of an architectural change, where the newly introduced *component C* overtakes the responsibility for providing the *interface S2*. Since the integration test model does not contain the new component, it is not anymore valid and must be modified. Namely, the *component C* with the *interface S2* should be added.

Once the test models are updated with the relevant changes coming from the system restructuring, they could be used in the last step, the sixth step, as input for the *forward engineering* of test cases, i.e., the actual test case generation using model-based testing.

To summarize, the overall aim is to provide the highest level of automation for each of the discussed steps, thus minimize required manual intervention in the semi-automated activities (cf. Fig. 1).

References

[1] H. M. Sneed, "Risks involved in reengineering projects," 1999.

[2] A. Jääskeläinen et al., "Synthesizing Test Models from Test Cases," 2009.

[3] A. Menychtas et al., "Artist methodology and framework: A novel approach for the migration of legacy software on the cloud," 2013.