Concept-based Co-migration of Test Cases

Ivan Jovanovikj, Enes Yigitbas, Stefan Sauer and Gregor Engels

Abstract: Software testing plays an important role in software migration as it verifies its success. As the creation of test cases is an expensive and time consuming activity, whenever test cases are existing, their reuse should be considered, thus implying their co-migration. During co-migration of test cases, two main challenges have to be addressed: situativity and co-evolution. The first one suggests that when a test migration method is developed, the situational context has to be considered as it influences the quality and the effort regarding the test case migration. The latter suggests that the changes that happen to the system have to be considered and eventually reflected to the test cases. We address these challenges by proposing a solution that applies situational method engineering extended with co-evolution analysis. The development of the test migration method is centered upon the identification of concepts describing the original tests and original system. Furthermore, the impact of the different realization of the system concepts in source and target environments is analyzed as part of the co-evolution analysis. Lastly, based on this information, a selection of suitable test migration strategies is performed.

1 INTRODUCTION

Software migration is a well-established method for transferring software systems in new environments while keeping the data and the functionality of the system (Bisbal et al., 1999). A widely used validation technique in software migration projects is software testing. As test case design, in general, has been seen as an expensive and time consuming activity (Sneed, 1999), reusing existing in migration context test cases is certainly worth considering. Reusing test cases cannot only substantially reduce the cost of testing the migrated system, but can also help to retain valuable information about the expected functionality of the original system and thus the desired functionality of the migrated system.

The migration of the test cases comes down to the problem of co-migration, i.e., the test cases have to be migrated along with the system as their migration is dependent on the system migration. The co-migration is practically defined by the co-evolution of the test cases and the corresponding system. In general, co-evolution refers to two or more objects evolving alongside each other, such that there is a relationship between the two that must be maintained (Mens and Demeyer, 2008). In our case, this refers to the test cases evolving alongside the code being migrated, such that the test cases remain correct for testing the migrated system. Hence, co-evolution analysis should be incorporated in the test case migration in order to provide a proper evolution of the test cases and thus, their proper migration.

When performing a test migration, a transformation method is required which serves as a technical guideline and describes the activities necessary to perform, tools to be used, and roles to be involved in order to migrate given test cases. The development of the transformation method is a very important task as it influences the overall success of the migration project in terms of effectiveness (e.g., non-functional properties of the migrated system) and efficiency (e.g., the time required or the budget). To achieve this, the situational context of the migration project should be taken into consideration. The situational context comprises different influence factors like characteristics of the original system or target environment, the goals of the stakeholders etc. Concerning test case co-migration, the situational context gets even more complex as beside the influence factors of the system migration, test-specific influence factors like characteristics of the original test cases or test target environment have to be considered as well. To develop a situation-specific transformation method is an important and challenging task, as the previously discussed co-evolution aspect should be incorporated when identifying the situational context from

Keywords: Test Case Migration, Co-migration, Co-evolution, Concept Modeling, Method Engineering.

Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved
In order to address the previously mentioned challenges, based on the Method Engineering Framework for Situation-Specific Software Transformation Methods (MEFiSTo) (Grieger, 2016), we provide a solution that combines techniques from Situational Method Engineering (SME) (Henderson-Sellers et al., 2014) and Software Evolution (Mens and Demeyer, 2008). Figure 1 depicts the overview of our method engineering process whose activities are split in two main disciplines: Method Development and Method Enactment. Beside the Method Engineering Process, another integral part of the solution approach is the Method Base. The Method Base contains the building blocks, Method Fragments and Method Patterns, needed for assembling the test migration method. Method Fragments are atomic building blocks of a migration method, whereas Method Patterns represent a proven migration strategy and indicate which fragments are necessary and how to assemble them together. The suitability of each pattern to a certain situation is expressed by a set of characteristics. Using the Method Base, the Method Engineering Process guides the development and the enactment of the situation-specific test migration method. By performing activities of the Method Development discipline, a situation-specific test method gets developed. It comprises the following two activities: Situational Context Identification and Transformation Method Construction. As our focus is on the incorporation of the co-evolution analysis in the method engineering process, in this paper, we mainly focus on the Situational Context Identification activity. During this activity, the situational context is analyzed and characterized from both system and test perspectives. In Section 2, we introduce the running example that we use throughout the paper. In Section 3, we present the identification of the concepts. In Section 4, the co-evolution analysis is explained. The influence factor identification is introduced in Section 5. In Section 6, we discuss related work and at the end, in Section 7, we conclude our paper and give an outlook on future work.

2 RUNNING EXAMPLE

As a running example, we use a real-world migration project (Schwichtenberg et al., 2018), where the problem of enabling cross-platform availability of the well-known Eclipse Modeling Framework (EMF) was addressed. EMF is highly adopted in practice and generates Java code from platform independent models with embedded Object Constraint Language (OCL) expressions. EMF provides an implementation of OCL which is an OMG standardized formal lan-
language used to describe expressions over UML models. As feature-complete Ecore and OCL runtime APIs are not available for some platforms, their functionality has to be re-implemented for each of them. The EMF Code Generator component embeds native string-based OCL expressions directly in the emitted Java code with the help of Java Emitter Templates. The interpretation of the OCL expressions is delegated by the EMF Runtime API to the OCL Interpreter. The interpreter firstly parses the string-based OCL expressions and evaluates them at run-time, i.e., in Just-In-Time (JIT) manner.

CrossEcore (Schwichtenberg et al., 2018), a cross-platform enabled modeling framework, addresses this problem by providing a code generation of platform-specific code from platform-independent Ecore models with embedded OCL expressions. The OCL compiler (OCL Visitor), as part of the CrossEcore Code Generator, transcompiles the string-based OCL expressions into expressions of the target programming language. Hence, the OCL expressions are translated at design-time and ahead of compilation, i.e., in Ahead-Of-Time (AOT) manner.

As EMF’s OCL implementation is well-tested, with more than 4000 JUnit test cases being available on public code repositories¹, their reuse was a very intuitive solution to be selected. However, their reuse was not that straightforward as CrossEcore’s OCL implementation was completely different in comparison to EMF’s OCL implementation. As the migration was performed to different target platforms, i.e., programming languages, a “one-size-fits-all” approach is not the perfect solution. This implies usage of a situation-specific transformation method, for example, suitable for the situation characterized by the target language or the target testing platform. This example scenario is used throughout the paper to present each of the activities of the situational context identification.

3 CONCEPT IDENTIFICATION

The purpose of the Concept Identification activity is to model a decomposition of the system and the testing artifacts into distinct parts for both the source and the target environment. We use the principles of Concept Modeling (Kozaczynski et al., 1992) to represent the functionality of the system as a set of concepts. The concepts are split into two different groups, namely language concepts and abstract concepts. Language concepts directly correspond to syntactic entities of the programming language, like variables, declarations, statements etc. (Kozaczynski et al., 1992). The abstract concepts, on the contrary, represent language-independent ideas of computation and problem solving methods (Kozaczynski et al., 1992). Abstract concepts are further classified into architectural and programming concepts. The architectural concepts are associated with interfaces or components whereas the programming concepts represent a general coding strategy, data structure or algorithm. Concepts can be related to each by is-a relation, to express a hierarchy between concepts, and consists-of relation to express dependencies between concepts. In (Grieger, 2016), when applying the idea of Concept Modeling to software modernization, three classes of concepts are distinguished: original system concepts, target system concepts, and shared system concepts. Regarding the original system, the language concepts are determined by the language elements that are already used, whereas language concepts regarding the target system concepts are those language concepts that will be used after the transformation. Finally, a shared concept is an abstract concept of the original system that can be realized in the target environment. All in all, the concept model is defined as a directed, acyclic and connected graph. The nodes represent the concepts, whereas the edges between them represent is-a or consists-of relations. In the following, we present the concept model as well as the concept identification process.

As shown in Figure 2, our Concept Model is a part of the Situational Context Model. The Concept Model, which extends the concept model introduced in (Grieger, 2016), can contain a set of Concerns which can further contain sub-Concerns and a set of Concepts. Beside the SystemConcept subclass which expresses system related concepts, the ConceptModel contains an additional subclass for expressing test-related concepts, namely the TestConcept class. This class has additional subclasses like the AbstractTestConcept which is further specialized into ProgrammingTestConcept and ArchitecturalTestConcept, and the LanguageTestConcept for expressing concrete syntactic entities related to testing. As defined by the conceptClass attribute in the Concept class, each Concept belongs to one out of six classes as defined by the enumeration type ConceptClass. Furthermore, a Concept can be related to other concepts by the consists-of and is-a relations. The invariants of the concept model instances are ensured by the OCL constraints. For example, the two OCL constraints shown in Figure 2 define that the target of a consists-of relation for a SystemConcept or a TestConcept can only be another SystemConcept or ProgrammingConcept, respectively.

¹http://git.eclipse.org/c/ocl/org.eclipse.ocl.git/tree/tests/
The concept identification process defines the necessary steps to be performed in order to capture the test and system concepts for both source and target environments. The process is target-driven as the desired transformation outcome is to identify and deliver the concepts to be identified. Firstly, a set of concerns is identified based on system and test target architectures, which in turn provides a coarse-grained structure of the concept model. The next three activities (A - C) are performed repeatedly for each concern. Firstly, concepts for the current concern are identified based on the target system and the target tests (A). This activity is based solely on the experience of the expert or by evaluating supporting materials like development or test tutorials. Figure 3 shows the concept model of the example introduced in Section 2. From system perspective, we have identified the abstract architectural concept OCL (Ahead-of-Time) which represents the realization of the shared abstract concept Object Constraint Language (OCL) in EMF. It consists of a concrete programming concept named Language-specific OCL-Expression, which represents the OCL expression that is defined in CrossEcore by using language constructs. This class consists of Derived Expression, Operation, and Constraint. The first two programming concepts consist of Functions, whereas the last one is a Logical Expression. From testing perspective, we have identified the OCL Test Case as shared abstract test concept. As a target test concept we have identified the language test concept OCL Test Case (AOT), which further consists of the language test concepts Action, Expected Result, and Assertion. Secondly, by performing a superficial analysis of the original system and original tests the identified concepts are validated (B). This is necessary, because the concepts have so far been identified without considering the original system and the original tests. Thirdly, a concluding source-driven identification takes place as original system and original tests are analyzed to eventually identify additional concepts that are specific to the original system and the original test cases (C). From system perspective, we have identified the abstract architectural concept OCL (Just-in-Time) which represents the realization of the shared abstract concept Object Constraint Language (OCL) in EMF. It consists of a concrete programming concept named Native OCL-Expression which represents the OCL expression in EMF which further consists of Derived Expression, Operation, and Constraint all of them, as typical for EMF, being String Literals. From testing perspective, we have identified the source test concept OCL Test Case (AOT). This concept consists of Assertion, which further consists of the two language test concepts Action and Expected Result.

4 CO-EVOLUTION ANALYSIS

Having identified the concepts, from both system and test perspective, in terms of a concept model, in this step a co-evolution analysis is performed. According to (Mens and Demeyer, 2008), the co-evolution process consists of several activities from which Change Detection and Impact Analysis are relevant during the situational context identification. During Change Detection, all changed parts of the system being mi-
grated are identified. Having identified these changes, all affected parts of the test cases are identified in the next step called Impact Analysis. Finally, an estimate of the effort required to accomplish the changes together with involved risks is made. Therefrom, the main idea is to describe the changes that happened to the system by identifying and relating corresponding source and target system concepts to each other.

Then, the relation between the source and target test concepts to their corresponding system concepts is also established. Having these relations, the impact of the system changes on the test cases can be derived. In order to enable these activities, we provide a metamodel (shown in Figure 4) that formalizes the relations between the system and test concepts in terms of traces. Furthermore, we also introduce the impact analysis process which defines the necessary actions to perform the impact analysis.

As can be seen in the upper left of Figure 4, we consider the Impact Model to be a part of the SituationalContextModel. The ImpactModel can contain a set of Traces which can be either CorrespondsToTrace or a ContainsTrace. Each Trace has a source and a target Concept. In the case of a CorrespondsToTrace, as defined by the related OCL constraint, both target and source are of the type SystemConcept. On the other hand, a ContainsTrace, as defined by the related OCL constraint, has as target a TestConcept and a SystemConcept as source.

As previously introduced, the co-evolution analysis addresses two main activities, namely, the detection of the changes and the impact analysis. Firstly, on the basis of the input concept model, for each source system concept, a corresponding target system concept is identified and a CorrespondsToTrace is created. Then, each test concept, both source and target, is related to the corresponding system concept that it tests (contains) by establishing a ContainsTrace. Af-
fter the second activity is done, a complete traceability model is produced in terms of an ImpactModel. The traces in the ImpactModel express the actual impact. Related to the example shown in Figure 3, the source system concept Native OCL-Expression corresponds to the target system concept Language-specific OCL-Expression. Regarding the test concepts, on the one hand, the source system concept Native OCL-Expression is contained by the source test concepts Action and Assertion. On the other hand, the corresponding system target concept Language-specific OCL-Expression is contained by the target test concepts Action, Expected Result, and Assertion. These relations suggest indirectly in which way the tests are influenced by the system changes. Namely, the test cases have to be changed in a way that the contained part of the system should be changed in accordance to the correspondence relation between the system concepts Native OCL-Expression and Language-specific OCL-Expression.

5 INFLUENCE FACTOR IDENTIFICATION

In this activity, similarly to (Grieger, 2016), for each identified concept, a method pattern is chosen. In order to decide which test method pattern to use, one needs to systematically search for characteristics that influence pattern’s efficiency or effectiveness. Each pattern has a set of characteristics which actually express its suitability to a certain situation. To support the influence factor identification, we provide an influence factor metamodel and a general guideline for identifying influence factors.

The InfluenceFactorModel is a part of the SituationalContextModel (Figure 5). It can contain a set of InfluenceFactors, further split into two subclasses: TestInfluenceFactor and SystemInfluenceFactor. An influence factor is defined as a characteristic of a co-migration project that has some impact on the efficiency or effectiveness of a transformation method. The TestInfluenceFactor is used to describe the test-related influence factors for both source and target environments as well as organizational and test tooling-related influence factors. Similarly, one can use the SystemInfluenceFactor to specify influence factors from system perspective. Note that an additional class appears, namely the TransformationInfluenceFactor, which has the role to specify the influence that the system migration could eventually have on the test case migration. A given Concept is associated with a set of suitable MethodPatternAlternatives, showing that each MethodPatternAlternative can be influenced by InfluenceFactors. Then, one pattern can be chosen to be applied, which is expressed by the AppliedMethodPattern class. For both MethodPatternAlternative AppliedMethodPattern classes, we distinguish between test and system migration patterns, indicated by the subclasses SystemMethodPatternAlternative and TestMethodPatternAlternative, and AppliedSystemMethodPattern and AppliedTestMethodPattern, respectively. For example, the realization of a concept in the original system will influence all suitable method patterns. However, the InfluenceFactor only needs to be specified once and can be linked to all affected MethodPatternAlternatives. In order to ensure invariants in the model, OCL constraints are used. For example, the OCL constraint related to the class TestInfluenceFactor defines that each TestInfluenceFactor can influence only a TestMethodPatternAlternative.

In the following, we describe the necessary steps for instantiating of an influence factor model. Firstly, a fine-grained analysis of the test and system realization, both source and target, for each concept is performed. In contrast to the coarse-grained superficial analysis from context identification, here we identify influence factors that may allow exclusion of some of the possible test method patterns, thus reducing the overall analysis effort. For example, if the realization of the given concept in the original and the target environment is significantly different, the patterns which do not include conceptual transformation are not suitable. Once a set of suitable test method patterns is obtained, influence factors for each pattern are identified and described. In order to systematize the process of identifying influence factors, each test method pattern is analyzed from the perspective of the comprising method fragments. For example, as illustrated in the third pattern (from left to the right) in Figure 6, a method fragment defines that a parser is needed. In this case, the availability of such a parser is an influence factor. The situational context model contains the identified abstract test concept, namely the OCL Test Case, its realization in the source test environment as well as the planned realization in the target test environment. Additionally, it also contains the assessed suitability of the possible test method patterns, in terms of effectiveness and efficiency. For example, regarding the leftmost method pattern, the testers would be challenged with re-implementing the tests cases as they are not experienced with CrossEcore. The second pattern has been identified as not suitable as it does not provide a way to interact directly with the test concepts like expected result or the test action. Having the influence factors identified, all suitable test method patterns are analyzed and assessed.
6 RELATED WORK

Different categories of method engineering approaches exist, namely those that provide fixed methods, a selection out of a set of fixed methods, configuration of a method, tailoring a method or a modular construction of the method (Grieger, 2016). Here, we only consider the last two, as they provide higher level of flexibility. The method tailoring approaches enable tailoring of a provided method, which can be changed arbitrarily (REMICS (Mohagheghi, 2010), SOAMIG (Zillmann et al., 2011), ARTIST (Menychtas et al., 2014). The problem is, however, that the process of changing the provided method is not guided by a method engineering process. The approaches that provide modular construction of a method rely on a set of predefined building blocks for methods and a method engineering process that guides the method construction. A method engineering approach that enables modular construction is presented in (Khadka et al., 2011), but is specific for migration to service-oriented environments. The MEFiSTo framework (Grieger, 2016) overcomes this issue by providing a general solution for modular construction of situation-specific migration methods. However, all of these methods do not provide support for migration of test cases (except ARTIST (Menychtas et al., 2014) to some extent), namely the consideration of the test context, as well as the analysis of the impact that the system changes have on the test cases. In the area of test case evolution, work is predominantly oriented to the continuous evolution of test cases with the system. Compared to the evolution of test cases in a migration setting, it is much more fine-grained. In (Farooq et al., 2010), a model-based regression testing approach is proposed for evolving systems with flexible tool support. However, this method does not address the conceptual changes in the system migration as well as their impact on the
test cases. (Mirzaaghaei et al., 2012) provides a semi-automatic approach that supports test suite evolution through test case adaptations by automatically repairing and generating test cases during software evolution. This approach deals with the problem of repairing existing test cases and generating new ones to react to incremental changes in the software system. Lastly, (Rapos, 2015) proposes a method for improving the model-based test efficiency by co-evolving test models alongside system models. To enable this, software model evolution patterns and their effects on test models are analyzed. As all these approaches deal with incremental changes, none of them is able to detect conceptual changes.

7 CONCLUSION AND FUTURE WORK

In this paper, we presented a method engineering process that enables a modular construction of situation-specific test migration methods. The process consists of two main disciplines, namely method development and method enactment, and relies on a method base. The focus in this paper is on the method development, particularly on the situational context identification. Firstly, the identification of system and test concepts realized in both source and target environments is performed. Then, the different realization of the system and test concepts is analyzed as part of the co-evolution analysis. Finally, test and system influence factors are identified and, based on this information, a selection of a suitable test migration strategy is performed. As future work we plan to develop tool support for the activities of the proposed approach, e.g., to support the modeling of the situational context. Furthermore, automation of the impact analysis based on the obtained concept models is also planned. Additionally, a quality analysis of the constructed test migration methods regarding quality criteria like completeness or correctness is planned.

REFERENCES


