Test Case Co-Migration Method Patterns

Ivan Jovanovikj, Enes Yigitbas, Stefan Sauer, Gregor Engels
Software Innovation Lab
Paderborn University, Paderborn, Germany
Email: firstname.lastname@uni-paderborn.de

CEUR-WS.org/Vol-2581/emls2020paper2.pdf

***Abstract*—Co-migration of test cases has a twofold benefit: it reduces the cost of testing the migrated system and retains valuable information about the expected functionality of the original system and thus the desired functionality of the migrated system. The migration of test cases is shaped by the co-evolution of the test cases, as they can be affected by the changes in the system migration. Furthermore, the situational context has to be considered as it influences the quality and the effort regarding the test case migration. To address these challenges, we propose a solution that applies situational method engineering extended with co-evolution analysis. The proposed framework enables modular construction of test transformation methods which consists of a method base and a method engineering process. Method fragments are the atomic building blocks of a migration method, whereas method patterns encode specific migration strategies. Beside the basic test method patterns, we introduce co-migration patterns, which encode the dependency between the system migration and the test case migration. The method engineering process provides the guidance on development and enactment of migration methods. In this paper we give an overview of the method base, in particular on the co-migration method patterns, as well as a detailed discussion.**

***Index Terms*—test case migration, co-migration, co-evolution, method engineering, method-base, co-migration method pattern**

I. Introduction

Reusing existing test cases is a frequently used validation strategy in software migration [1]. It can reduce the cost of testing the migrated system and can also help to retain valuable information about the expected functionality of the original system and thus the desired functionality of the migrated system. Reusing test cases comes down to the problem of co-migration, i.e., the test cases have to be migrated along with the system to the dependency 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 [2]. In our case, this refers to the test cases evolving alongside the migrating code, such that the test cases remain correct for testing the migrated system. This implies that the co-evolution analysis should be incorporated in the test case 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) 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 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 both system and test perspective.

In order to address the previously mentioned challenges, based on the *Method Engineering Framework for Situation-Specific Software Transformation Methods (MEFiSTo)* [3], we provide a framework that combines techniques from *Situational Method Engineering (SME)* [4] and *Software Evolution* [2]. In general, a *Method Base* contains the building blocks needed for assembling the migration method, namely *Method Fragments* and *Method Patterns*. A *Method Fragment* is an atomic building block of a migration method, whereas a *Method Pattern* represents a strategy and indicates which fragments are necessary and how to assemble them together. As the test method patterns cannot express directly the dependency between the system and the test case migration, we propose a set of co-migration method patterns. Technically, a co-migration method pattern is a combination of a system method pattern and a test method pattern visually resembling to a double horseshoe model. A co-migration pattern encode the relation between the applied system migration pattern and the selected test method pattern.

The structure of the rest of the paper is as follows: In Section II, we introduce the test method fragments. Then, in Section III, we introduce the test method patterns. In Section IV, we present the test co-migration method patterns. In Section V, we briefly discuss the related work and at the end, Section VI concludes the work and gives an outlook on future work.

Copyright © 2020 for this paper by its authors.Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

II. Method Fragments

A method fragment is an atomic building block of a migration method, i.e., an activity, artifact or tool. As we follow the idea of model-driven software migration [5], our method fragments belong to one of the following reengineering processes: *Reverse Engineering*, *Restructuring*, and *Forward Engineering* [6]. The activities as well as the artifacts are represented in Figure 1 as an instance of the well-known horseshoe model [7].

Artifacts are constituting parts of each migration method and are distinguished by the level of abstraction they are belonging to. On the *System Layer*, textual artifacts representing test code and models of the test code are placed. Regarding the textual artifacts, this is either the *Original Test Code* or the *Migrated Test Code*. Similarly, regarding the models of the code it is either the *Model of Original Test Code* or the *Model of the Migrated Test Code* represented in a form of in a form of an Abstract Syntax Tree [8]. The *Platform-Specific Layer* is a higher level of abstraction compared to the system layer. Here, technology-specific concepts are used to represent the test cases for both the source and the target environment in terms of *Model of Original/Target Executable Tests*. On the *Platform-Independent Layer* the models representing the test cases are independent of any particular testing framework or testing technology. Here, we distinguish between two types of models, the *Model of the Abstract Tests* and the *System Behavior Model*. On the highest level of abstraction, we foresee the *System Behavior Model* that represents a compact representation of the expected behavior of the system. Behavioral diagrams like the UML activity or sequence diagram, or state machines can be used to represent the expected behavior of the system.

Activities in the test case reengineering horseshoe model produce or consume appropriate artifacts. As can be seen in Figure 1, these activities can be distinguished by the reengineering process they belong to, namely *Reverse Engineering*, *Restructuring*, or *Forward Engineering*.

Figure 1. Excerpt of Method Fragments stored in the Method Base.

Reverse Engineering, is the process of analyzing a test case and creating another representation of them on a higher level of abstraction, e.g., by using test models. In general, reverse engineering can be seen as combination of *Model Discovery* and *Model Understanding* [9]. The *Model Discovery* step relies on syntactical analysis and by using a parser it allows automatic text-to-model transformation to create a model of the test case source code represented as an Abstract Syntax Tree (AST) [8]. *Model Understanding*, in general, is a model-to-model transformation activity, or a chain of such activities (*Test Case Understanding*, *Test Abstraction*, and *System Behavior Recovery*), which takes the initial models, applies semantic mappings and generates derived models on a higher level of abstraction.

Restructuring, is defined as the transformation from one test representation to another at the same relative abstraction level, while preserving the functionality that is being checked by the tests. This activity has been foreseen on both the *System Behavior Model* and the *Model of Abstract Test*. The *Restructuring* activity is of course influenced by the target testing environment, testing tool, or by requirements on improving the quality of the test cases (e.g., maintainability). Furthermore, it could also be influenced by the changes that happen in the system migration.

Forward Engineering, is the process of moving of high-level test abstractions and logical implementation-independent design to the physical implementation of the test cases. The test models are used as input for a chain of model-to-model transformations (*Abstract Test Derivation* and *Test Concretization*), ending with a model-to-text transformation (*Test Code Generation*), which provides the test code as output.

The *Reimplementation* is a text-to-text transformation which is performed manually. The *Language Transformation* also known as AST-based transformation [7], defines a direct mapping between the *Original Model of the Test Code* and the *Migrated Model of the Test Code*. The *Test-Language Transformation*, on the other hand, defines a mapping directly between two testing frameworks. The *Enrichment* activity is applicable to various models, e.g., *Original Model of Executable Tests*, *Model of Abstract Tests*, or *System Behavior Model*. The *Removal* activity is used to specify which part of the test case code should not be transformed.

III. Method Patterns

The method fragments like artifacts, activities or tools are not sufficient as no guidance is provided how to assemble them. A method pattern, represents construction guidelines for migration methods that follows a certain strategy by defining which method fragments should be customized and how to put them together In the following, as shown in Figure 2, we present an excerpt of the method patterns that preserve functionality.

The *Language-based Test Transformation* pattern defines the migration of the functionality of the test cases by defining a mapping between the language constructs in both original and target environment. The mapping is applied by a direct

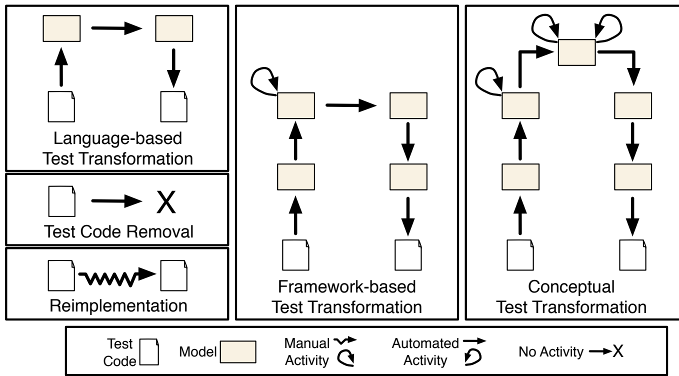

Figure 2. Excerpt of basic method patterns.

transformation between *Model of the Original Test Code* and *Model of the Migrated Test Code*. Theoretically, this pattern could be applied actually in any migration scenario, but its suitability mainly depends on the complexity of the model transformations between both models. From test perspective, the transformation of the test concepts have to be done implicitly.

By using the *Test Language-based Test Transformation* pattern the functionality of the test cases is migrated by using an intermediate test representation on platform-specific layer. *Model of Original Executable Tests* represents explicitly the testing constructs and the test data. Doing so, transformation step is less complex compared to the *Language-based Test Transformation* and it enables a direct, i.e., an explicit representation and manipulation of test constructs.

The *Conceptual Test Transformation* pattern defines to migrate the test functionality by using an intermediate representation in terms of *Model of Abstract Tests* on a platform-independent layer. This improves the dependent framework transformation on the platform-specific layer by explicitly representing some test concepts on a higher level of abstraction as part of the *Model of Abstract Tests*. This pattern could be considered suitable when some test concepts are realized completely different in both environments or when a restructuring of the test architecture or test data is necessary. By using the *Reimplementation* pattern the functionality of the test cases is manually transformed by software developers and it is suitable in cases when an automatic migration is too complex to be implemented. Lastly, the *Test Code Removal* pattern defines not to migrate certain part of the test code, e.g., when some parts of the original system are now implicitly supported in the new environment.

IV. Co-Migration Method Patterns

As the test method patterns cannot express the relation between the system and the test case migration, we propose a set of co-migration method patterns. Technically, a co-migration method pattern is a combination of a system method pattern and a test method pattern, visually resembling to a double horseshoe model. We define a co-migration method pattern as follows:

A co-migration method pattern is a method pattern which relates a test method pattern and a system method pattern by explicitly establishing the relation between the corresponding method fragments.

By explicitly establishing the relation between test and system method patterns, we aim to ease the process of the selection and configuration of a test method pattern. An already configured system method pattern, with selected and concertized method fragments, i.e., artifacts and activities, suggests in what way the test method fragments should be selected and configured. Consequently, it suggests in what way the tools supporting the different method fragments should be developed.

The co-migration patterns also facilitate reuse of existing artifacts and activities from the system migration method. As an explicit relation between the system and test method patterns exists, it facilitates the reuse of the already existing artifacts and activities defined for the system transformation method. Furthermore, the developed and used tooling for the system migration, e.g., a language parser or a language meta-model, which correspond to an activity or an artifact, could be reused.

In this work, as we already mentioned in Section III, we focus on the functionality preserving test method patterns. Our test method patterns were mainly inspired by the method patterns presented in [3], where four different functionality preserving method patterns were defined, namely: *Reimplementation*, *Language-based Transformation*, *Conceptual Transformation*, and *Code-Removal*. As we already said at the beginning of this section, a co-migration comprises a test method pattern and a system method pattern. We created the co-migration patterns by combining each of the test method patterns with each of the system method patterns, excluding the *Code-Removal* pattern. The *Code-Removal* was not taken into consideration as it has no influence on reuse of method fragments. In the following, we analyze each co-migration method pattern regarding two aspects reusable method fragments and impacted method fragments. Reusable method fragments are those method fragments from the system transformation method which could be directly reused in the test transformation method. Impacted method fragments are those test method fragments which are impacted from the system method fragments.

Figure 3 depicts the co-migration patterns that combine either a *Test Reimplementation* method pattern (*CMP1* to *CPM3*) or a *Language-based Test Transformation* method pattern(*CMP4* to *CPM6*) with the three possible system migration patterns *Reimplementation*, *Language-based Transformation*, and *Conceptual Transformation*.

The pattern *CMP1* is a combination of two reimplementation method patterns and it is a very simple pattern which suggests a manual migration of the test cases. The ease of reimplementation of the tests cases depends on the documentation of the system transformation method, the more structured the better. In the case of *CMP2* and *CMP3* patterns, where *Language-based Transformation* and *Conceptual Transforma-*

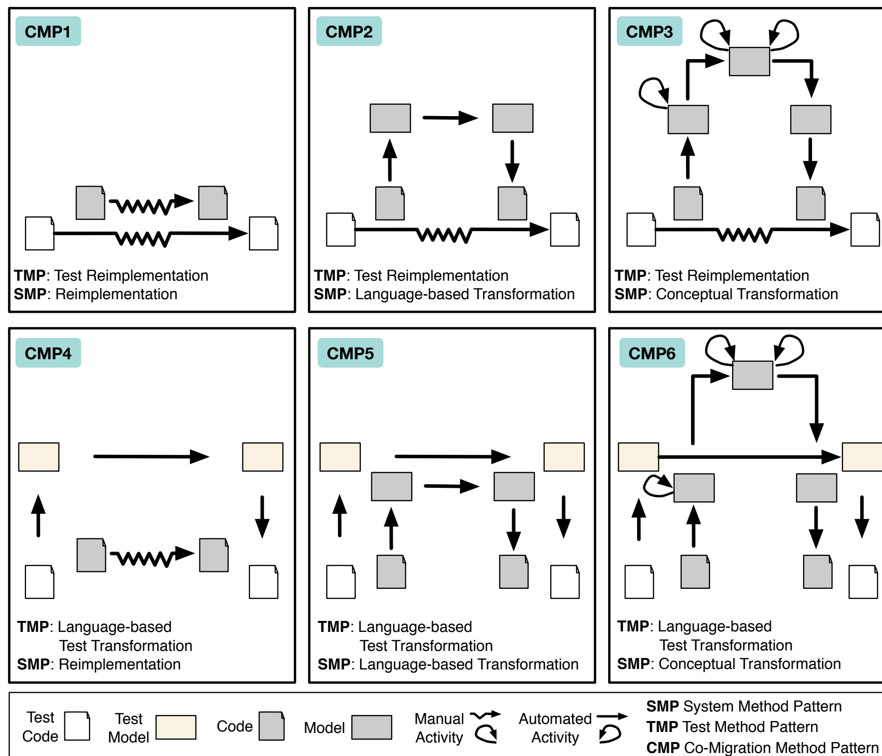

Figure 3. Co-Migration Patterns: Part I

tion are applied respectively, the reimplementation of the test cases should be easier as the transformation of the system is specified explicitly in terms of transformation rules. Basically, no system method fragments could be directly reused.

The pattern *CMP4* is a combination of a *Language-based Test Transformation* and a *Reimplementation*. This pattern is suitable if the system reimplementation was well documented so that some transformation rules can be derived in order to automate the transformation of the test cases. However, it suggests implementation of a parser for the source language as well as a code generator for the target language. Similarly to the previous co-migration patterns, no system method fragments could be directly reused.

The pattern *CMP5* is a combination of *Language-based Test Transformation* and *Language-based Transformation*. This pattern has a symmetric constellation, as two transformations on the same abstraction level are combined. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. Reuse of existing method fragments is also possible in the scope of the transformation step. However, the complexity of the transformation could be higher if the source and the target frameworks differentiate a lot, meaning that the transformation of the test relevant concepts should be done implicitly. Regarding the impacted test method fragments, the *Language Transformation* activity is impacted by the corresponding method fragment from the system transformation method. The pattern *CMP6* is a combination of *Language-based Test Transformation* and *Conceptual Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. Due to the difference in the abstraction levels, reuse of existing method fragments in the scope of the transformation step is only possible in an indirect way. Namely, the transformation on the conceptual level, could be used as an input when the language transformation of the test cases is performed, i.e., the conceptual transformation parameterizes the language-based test transformation. Similarly as with *CMP5*, the complexity of the transformation could be higher if the source and the target frameworks differentiate a lot due to the implicit transformation of the test concepts.

Figure 4, depicts the co-migration patterns that combine either a *Test Language-based Test Transformation* method pattern (*CMP7* to *CPM9*) or a *Conceptual Test Transformation* method pattern (*CMP10* to *CPM12*) with the three possible system migration patterns *Reimplementation*, *Language-based Transformation*, and *Conceptual Transformation*.

The pattern *CMP7* is a combination of a *Test Language-based Test Transformation* and *Reimplementation*. This pattern is suitable if the system reimplementation was well documented so that some transformation rules can be derived in order to automate the transformation of the test cases. However, it suggests implementation of a parser for the source language as well as a code generator for the target language. Furthermore, a test case understanding fragment and

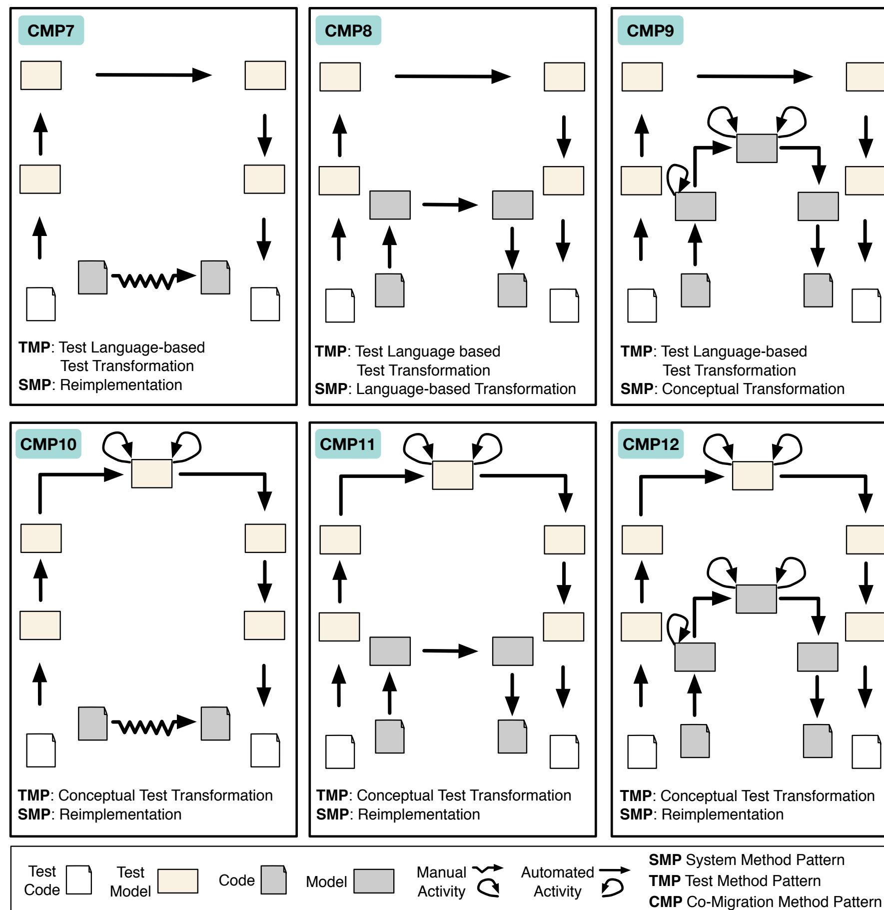

Figure 4. Co-Migration Patterns: Part II

test case concretiztaion fragment should be configured and implemented in terms of model-to-model transformations.

The pattern *CMP8* is a combination of *Test Language-based Test Transformation* and *Language-based Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be reused. Reuse of existing method fragments is also possible in the scope of the transformation step. But, a test case understanding fragment and test case concretiztaion fragment should be still selected and implemented in terms of model-to-model transformations. However, the complexity of the transformation is lower compared to *CMP7*, as the transformation activity from the system method pattern could be reused to higher extent as it is specified explicitly through a model-to-model transformation. On the other side, the complexity of the transformation is lowered as an explicit mapping between the testing languages is defined.

The pattern *CMP9* is a combination of *Test Language-based Test Transformation* and *Conceptual Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. Due to the difference in the abstraction levels, reuse of existing method fragments in the scope of the transformation step is only possible in an indirect way. Namely, the transformation on conceptual level could be used as an input when the *Test Language-based Test Transformation* is configured performed, i.e., the *Conceptual Transformation* parameterizes the Test *Language-based Test Transformation*. Similarly to *CMP8*, the complexity of the transformation is lowered as an implicit mapping between the testing languages is defined.

The pattern *CMP10* is a combination of *Conceptual Test Transformation* and *Reimplementation*. The suitability of this pattern depends on the system reimplementation, whether it was well documented so that some transformation rules can be derived in order to automate the transformation of the est cases. However, it suggests implementation of a parser for the source language as well as a code generator for the target language. Furthermore, a test case understanding fragment and test case concretiztaion fragment should be configured and implemented in terms of model-to-model transformations.

The pattern *CMP11* is a combination of *Conceptual Test Transformation* and *Language-based Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be reused. Reuse of existing method fragments is also possible in the scope of the transformation step. But, a test case understanding fragment and test case

concretiztaion fragment should be still selected and implemented in terms of model-to-model transformations. However, the complexity of the transformation is lower compared to *CMP10*, as the transformation activity from the system method pattern could be reused to higher extent as it is specified explicitly through a model-to-model transformation. On the other side, the complexity of the transformation is lowered as an explicit mapping between the testing languages is defined.

The pattern *CMP12* is a combination of *Conceptual Test Transformation* and *Conceptual Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. Due to the difference in the abstraction levels, reuse of existing method fragments in the scope of the transformation step is only possible in an indirect way. Namely, the transformation on conceptual level could be used used as an input when a *Test Language-based Test Transformation* method pattern is configured, i.e., a *Conceptual Transformation* pattern parameterizes the *Conceptual Test Transformation*. Similarly to *CMP11*, the complexity of the transformation is lowered as an implicit mapping between the testing languages is defined.

V. Related Work

Two main research areas are relevant for this work, namely method engineering and test case evolution. Regarding method engineering, different categories of method engineering approaches exist: fixed methods, a selection out of set of fixed methods, configuration of a method, tailoring a method or a modular construction of the method. The method tailoring approaches enable tailoring of a provided method, which can be changed arbitrarily ([10], [11]). However, they solely focus on system transformation and thus no transformation of test cases was considered. Consequently, they do not support comigration of test cases. The approaches that support modular construction of transformation methods provide a higher level of flexibility as they rely on a set of predefined building blocks for methods. The method engineering approach, presented in [12], enables modular construction, but is specialized for migration to service-oriented environments. This issue is addressed by the *MEFiSTO* Framework [3] by providing a general solution for modular construction of situation-specific migration methods. However, similarly to the method tailoring approaches, these two approaches do not address the migration of test cases (except ARTIST [11] to some extent) as well. The existing approaches in the test case evolution are predominantly focusing on the continuous co-evolution of test cases. In [13], a semi-automatic approach is presented that supports test suite evolution through test case adaptations. Existing test cases are repaired and new test cases are generated to react to incremental changes in the software system. In [14], a method is proposed which should improve the model-based test efficiency by co-evolving test models. As part of this work, software model evolution patterns as well as their effects on test models are studied in order to apply updates directly to the tests. All in all, the existing approaches deal only with incremental changes and not coarse grained changes, i.e., conceptual changes that often happen in software migration.

VI. Conclusion and Future Work

In this paper, we presented a framework that enables a modular construction of context-specific, model-driven migration methods for test cases. The framework consists of a method base and a method engineering process. The method base contains method fragments, as atomic building blocks of a migration method, and method patterns which encode specific migration strategies. In order to adequately address the co-evolution in test case co-migration, we propose a set of co-migration methods that encode the information about the dependency between the system and test case migration. In future work, we intend to conduct a quality analysis of the constructed test migration methods regarding quality criteria like completeness or correctness.

References

[1] J. Bisbal, D. Lawless, B. Bing Wu, and J. Grimson, "Legacy information systems: issues and directions," *IEEE Software*, 1999.

[2] T. Mens and S. Demeyer, *Software Evolution*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[3] M. Grieger, M. Fazal-Baqaie, G. Engels, and M. Klenke, "Concept-based engineering of situation-specific migration methods," in *Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness*, 2016.

[4] B. Henderson-Sellers, J. Ralyté, P. J. Ågerfalk, and M. Rossi, "Situational method engineering," in *Springer Berlin Heidelberg*, 2014.

[5] A. Fuhr, A. Winter, U. Erdmenger, T. Horn, U. Kaiser, V. Riediger, and W. Teppe, "Model-Driven Software Migration - Process Model, Tool Support and Application," in *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, 2012.

[6] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, 1990.

[7] R. Kazman, S. Woods, and S. Carriere, "Requirements for integrating software architecture and reengineering models: CORUM II," in *Proceedings Fifth Working Conference on Reverse Engineering*. IEEE Comput. Soc, 1998.

[8] OMG, *Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)- Version 1.0*. Object Management Group, 2011.

[9] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco," in *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*. ACM Press, 2010.

[10] C. Zillmann, A. Winter, A. Herget, W. Teppe, M. Theurer, A. Fuhr, T. Horn, V. Riediger, U. Erdmenger, U. Kaiser, D. Uhlig, and Y. Zimmermann, "The SOAMIG Process Model in Industrial Applications," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, 2011.

[11] A. Menychtas and E. Al., "Software modernization and cloudification using the ARTIST migration methodology and framework," *Scalable Computing: Practice and Experience*, 2014.

[12] R. Khadka, G. Reijnders, A. Saeidi, S. Jansen, and J. Hage, "A method engineering based legacy to soa migration method," in *27th IEEE International Conference on Software Maintenance (ICSM)*, 2011.

[13] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Supporting test suite evolution through test case adaptation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 231–240.

[14] E. J. Rapos, "Co-evolution of model-based tests for industrial automotive software," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–2.