Modular Construction of Context-Specific Test Case Migration Methods

Ivan Jovanovikj[1], Enes Yigitbas[1], Marvin Grieger[2], Stefan Sauer[1] and Gregor Engels[1]
[1]*Software Innovation Lab, Paderborn University, Paderborn, Germany*
[2]*VHV Gruppe, Hannover, Germany*

Keywords: Test Case Migration, Software Co-evolution, Software Reengineering, Method Engineering.

Abstract: Migration of test cases has a twofold benefit in software migration projects: reuse of valuable knowledge as well as time and cost savings. The diversity of software migration project contexts require a flexible and modular construction method to address several aspects like different system and test environments or the impact of the system changes on the test cases. When an inappropriate migration method is used, it may increase the effort and the costs and also decrease the overall software quality. Therefore, a critical task in test case migration is to provide a transformation method which fits the context. To address this problem, in this paper, we present a framework that enables a modular construction of context-specific migration methods for test cases by assembling predefined building blocks. Our approach builds upon an existing framework for modular construction of software transformation methods and 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. The guidance on development and enactment of migration methods is provided by the method engineering process. We evaluate our approach in an industrial case study where a part of the Eclipse Modeling Framework was migrated from Java to C#.

1 INTRODUCTION

Software testing plays an important role in software migration as it verifies whether the migrated system still provides the same functionality as the original system which is the main requirement in a software migration (Bisbal et al., 1999). Since software migration is established to reuse existing systems, we want to reuse test cases as well. The reuse of test cases 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 source system and therefore, about the functionality of the migrated system, too. A migration, in general, is performed by development and enactment of a migration method. A migration method, or a transformation method in a broader sense, defines which activities should be performed, which tools to be applied, and what artifacts should be generated in order to transfer a system from one environment to another. The same applies for the test case migration methods.

Having the right migration method and enacting it, is very important because a non-suitable migration method may lead to more complex and more expensive migration. Another aspect is the dependency between the system and the test cases. Since the test cases are coupled with the system they are testing, the system changes need to be detected, understood and then reflected on the test cases to facilitate reuse. In other words, the test cases need to be co-evolved. However, co-evolving test cases is far from being trivial since several challenges need to be addressed (Jovanovikj et al., 2016a), like quality assessment, refactoring or reflection of system changes. For this reason, we need a method which is capable of capturing such information, and accordingly enabling the development of a suitable test case migration method.

We address the aforementioned issues by providing a framework for modular construction of context-specific test case migration methods. Our approach is highly influenced by the MEFiSTO Framework (Grieger et al., 2016), as this approach provides high degree of controlled flexibility. We extend this framework by considering the test cases as well, thus enabling development of test case migration methods which enable automated co-evolution of test cases for a specific migration context. As shown in Figure 1, the solution approach consists of a *Method Base* and a *Method Engineering Process*. The *Method Base* contains the building blocks needed for assembling of the migration method, namely *Method*

Jovanovikj, I., Yigitbas, E., Grieger, M., Sauer, S. and Engels, G.
Modular Construction of Context-Specific Test Case Migration Methods.
DOI: 10.5220/0007690205340541
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 534-541
ISBN: 978-989-758-358-2

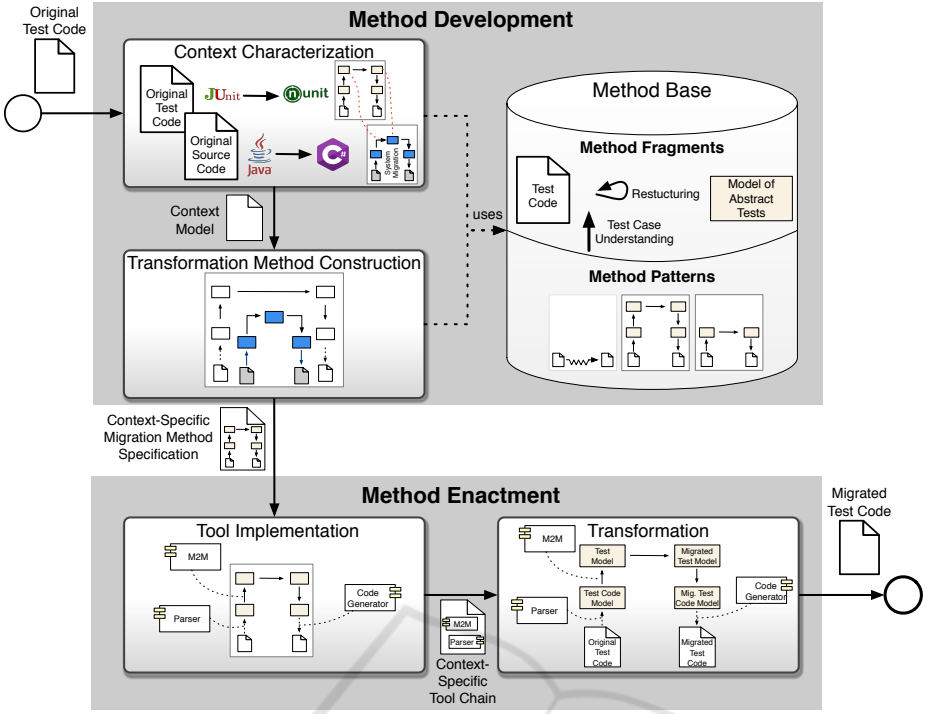

Figure 1: Approach Overview.

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. Having the *Method Fragments* and *Method Patterns*, a guidance is needed on how to create a migration method for a specific context. This is done by the *Method Engineering Process*, which guides the development and the enactment of the context-specific migration method.

The structure of the rest of the paper is as follows: In section 2, we introduce the content of the *Method Base*. Then, in section 3, we present the *Method Engineering Process*, which is instantiated in section 4 based on a case study from an industrial context. In section 5, we discuss the related work and at the end, in section 6 we conclude our paper and give an outlook on future work.

2 METHOD BASE

The *Method Base* contains the knowledge of the available migration strategies. It is actually a repository containing reusable building blocks of methods and includes method fragments and method patterns. In the following, we present an excerpt of the method fragments and patterns.

2.1 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 (Fuhr et al., 2012), our method fragments belong to one of the following reengineering processes: *Reverse Engineering*, *Restructuring*, and *Forward Engineering* (Chikofsky and Cross, 1990). The activities as well as the artifacts are represented in Figure 2 as an instance of the well-known horseshoe model (Kazman et al., 1998).

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*. The test code can be either the test cases, implemented in a specific testing framework, e.g. jUnit [1] or MSUnit [2], test configuration scripts or a manually implemented additional test framework.

Similarly, regarding the models of the code it is either the *Model of Original Test Code* or the *Model of the Migrated Test Code*. The *Model of the Test Code*

[1] http://junit.org/junit4/

[2] https://msdn.microsoft.com/en-us/library/ms243147.aspx

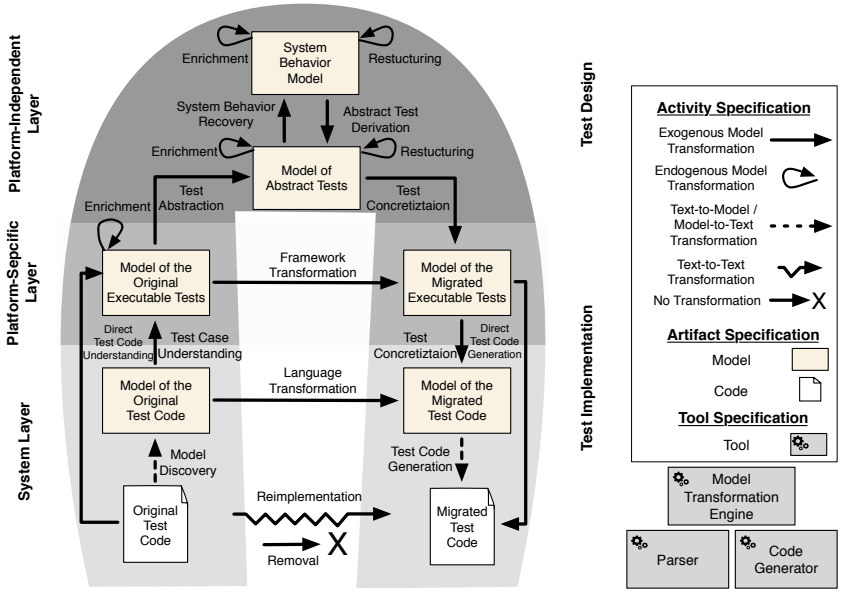

Figure 2: Excerpt of Method Fragments stored in the Method Base.

represents the test code in a form of an Abstract Syntax Tree (OMG, 2011) of the appropriate language of the original or the target environment.

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. *Model of Original/Target Executable Tests* is considered as platform-specific as it represents the executable test cases by using testing concepts which are specific for a particular testing framework, i.e., by using meta-models of jUnit or MSUnit to model the test code.

On the *Platform-Independent Layer* the models representing the test cases are independent of any particular testing framework or testing technology. On this level of abstraction, we distinguish between two types of models, the *Model of the Abstract Tests* and the *System Behavior Model*. The *Model of the Abstract Tests* is considered to be platform-independent as it is independent of any concrete testing framework. Standardized languages like UML Testing Profile (UTP) (OMG, 2013) and Test Description Language (TDL) (ETSI, 2016) are used for modeling the abstract tests. 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 2, these activities can be distinguished by the reengineering process they belong to, namely *Reverse Engineering*, *Restructuring*, or *Forward Engineering*.

Reverse Engineering, as defined in (Chikofsky and Cross, 1990), is the process of analyzing a subject system to create representations of the system in another form or on a higher level of abstraction. Applied in the software testing domain, it is a 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* (Bruneliere et al., 2010). 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) (OMG, 2011). *Model Understanding*, in general, is a model-to-model transformation activity, or a chain of such activities, which takes the initial models, applies semantic mappings and generates derived models on a higher level of abstraction. Here, we distinguish three sub-activities: Firstly, the initial models are explored by navigating through their structure in an activity called *Test Case Understanding*. Model elements which represent test relevant concepts like test suite, test case or assertion are identified and then, by applying a model-to-model transformation, a test model of executable test cases is obtained which is platform specific and it is an instance of a metamodel of a particular testing framework (e.g., jUnit, MSUnit or TTCN-3). By applying the *Test Abstraction* activity, which is a model-

to-model transformation as well, one can obtain a model of the abstract test cases which is platform-independent. The *System Behavior Recovery* activity is applied in order to obtain the highest possible level of abstraction defined by our reengineering model.

According to (Chikofsky and Cross, 1990), *Restructuring* is "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)". In the testing domain, we define test restructuring as the transformation from one test representation to another at the same relative abstraction level, while preserving the "semantics" of the tests. Here, with "semantics" we mean the functionality that is being checked by a particular test. 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). However, it could also be influenced by the changes that happen in the system migration. Since these changes may be relevant for the test models, they have to be reflected on the tests as well.

As defined by (Chikofsky and Cross, 1990), *Forward Engineering* is "the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system". In the field of software testing, this can be paraphrased as a 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, ending with a model-to-text transformation, which provides the test code as output. *Abstract Test Derivation* takes as input the *System Behavior Model* and produces *Model of Abstract Test Cases*, a platform-independent model of the test cases. Then, by applying *Test Concretization* a *Model of Migrated Executable Test Cases* is obtained. This model is already specific for a particular testing framework and can be used for the generation of the *Test Code* by executing the *Test Code Generation* activity, which is a model-to-text transformation. Since test cases are generated from the test model, this can be seen as e typical model-based testing activity (Jovanovikj et al., 2016b). The tools which support forward engineering are known as generators and for each specific target platform, an appropriate generator is needed. Custom code generators can be built by using Xtend [3] which is a statically typed programming language which offers features like template expressions and intelligent space management. The *Reimplementation* is a text-to-text transformation which is performed manually The *Language Transformation* also known as AST-based transformation (Kazman et al., 1998), defines a direct mapping between the *Original Model of the Test Code* and the *Migrated Model of the Test Code*. The *Framework 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.

In order to support the previously introduced activities, thus enabling a (semi-)automatic transformation, we foresee in total three types of tools that are needed. Firstly, a *Parser* is needed to obtain the initial models out of the textual artifacts, i.e, to perform the *Model Discovery* activity. Then, in a series of model-to-model transformations, which are specified by transformation rules, initial models are obtained. The specified rules are then executed by an *Model Transformation Engine*. Finally, a *Code Generator* is needed to perform the model-to-text transformation by executing the test code generation rules previously specified, thus generating the test code for the migrated test cases.

2.2 Method Patterns

Having only method fragments like artifacts, activities or tools is not sufficient as no guidance is provided how to assemble them and thus create a test case migration method. For this purpose, we additionally provide method patterns.

A method pattern, intuitively, represents construction guidelines for migration methods and follows a certain strategy. It contains the methodological knowledge with the purpose to address the problem associated to it. Technically seen, a method pattern defines which method fragments should be customized and how to put them together. Functionality preservation is a main requirement in any migration project, and therefore, in test case migration too (Bisbal et al., 1999). To preserve this functionality, a consistent path in the horseshoe model has to be realized from the *Original Test Code* to the *Migrated Test Code*. In the following, as shown in Figure 3, we present an excerpt of the method patterns that preserve functionality. Furthermore, we have also observed architectural restructuring pattern, which provide a methodological solution to make a architectural changes in the test cases during their migration. Due to space constraints, we only discuss the former cate-

[3] http://www.eclipse.org/xtend/

gory of migration method patterns.

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 transformation between the *Model of the Original Test Code* and the *Model of the Migrated Test Code*. Theoretically seen, 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. Firstly, the extracted *Model of the Original Test Code* needs to be interpreted to identify the test concepts to be transformed. Then, it could be necessary to restructure the explicit test concepts, like test behavior or test assertions. Once prepared, the test concepts on the original environment have to be mapped to the test concepts of the target environment. Thus, this pattern could be considered suitable if the effort of test interpretation and restructuring is relatively low. However, from test perspective, it basically means, the transformation of the test concepts have to be done implicitly.

The *Framework-based Test Transformation* pattern defines to migrate the functionality of the test cases by using an intermediate test representation on platform-specific layer. The testing concepts together with the test data are explicitly represented by a *Model of Original Executable Tests*. By applying the separation of concerns principle, this pattern makes the transformation step less complex compared to the *Language-based Test Transformation*. Namely, the first concern of interpretation, is explicitly addressed by the *Test Case Understanding* activity. Then, the *Restructuring* could be applied on the *Model of Original Executable Tests*, which enables direct manipulation of the test concepts, e.g. test assertions. After *Restructuring*, the mapping to the target test framework, i.e., mapping to the *Model of Migrated Executable Tests* is performed. Finally, *Direct Test Code Generation* is applied either directly or via *Test Concretization* and *Model of the Migrated Test Code*, the *Migrated Test Code* is generated. This pattern could be considered suitable when the difference of the test case implementation is significantly different in the original and the target testing frameworks. Compared to the *Language-based Test Transformation* pattern, this pattern enables direct, i.e., 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 a *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*. For example, from the test design perspective, the test architecture or test behavior could be explicitly represented with this model. 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.

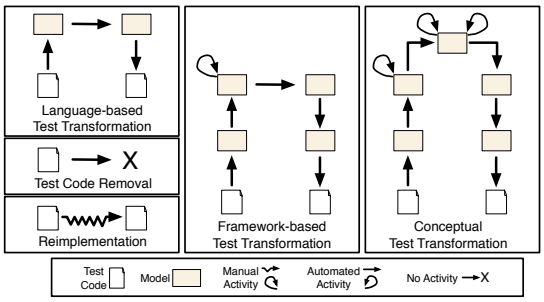

Figure 3: Excerpt of basic method patterns.

The *Reimplementation* pattern defines to migrate the test functionality by having it manually transformed by software developers. The main task is to implement the same test cases, testing the same functionality in the target environment. This pattern could be suitable in cases when an automatic migration is difficult to be implemented. The *Test Code Removal* pattern defines not to migrate certain part of the test code, i.e., no transformation should be performed on it. For example, it could be that some parts of the original system are now implicitly supported in the new environment, e.g., by a library or a framework, so there is no longer need to be tested.

3 METHOD ENGINEERING PROCESS

The method engineering process describes the main activities to be followed in order to create a context-specific test case migration method as well as their relation to the method base. As already shown in Figure 1, the activities are split in two main disciplines: *Method Development* and *Method Enactment*. The essential process input is the *Original Test Code*, which gets transformed to *Migrated Test Code* once the process is being enacted. The *Migrated Test Code* represents the test cases which can be run in the target environment and validate the system migration. The activities of both disciplines are associated with a flow that is attached with a *Context-Specific Migration Method Specification* which describes the actual migration method.

3.1 Method Development

By performing activities of the *Method Development* discipline a context specific method gets developed. The main activities are *Context Characterization* and *Transformation Method Construction*.

The first activity is the *Context Characterization*, in which the migration context is analyzed and characterized, from both system migration and testing perspective. Furthermore, impact analysis is also performed to identify the impact that the system changes have on the test cases. The gathered knowledge about the migration context is required in order to develop a suitable test case migration method. In order to structure the context information, we developed a migration context model, whose excerpt is shown in Figure 4. It mainly contains the technical background of the migration context, i.e., the characteristics of the source environment of the system and the test cases, the target system environment, and the desired test target environment. Last but not least, the system transformation characteristics are also analyzed as they are of great importance to understand their impact on the test cases. For example, the kind of system transformation or the target testing environment are important context characteristics which should be considered. Another purpose of the this activity is to perform an initial assessment of the method patterns against a set of influence factors. On the basis of information in the obtained context model a set of influence factors are set. Then, each pattern is analyzed against each identified influence factor, which in turn could have either positive or negative influence on the selection of a particular pattern.

Having the context information collected, the *Transformation Method Construction* activity can be initiated. In this activity, on the basis of the previously identified context information, a context-specific migration method gets constructed. First of all, a suitable pattern is selected, which in turn means a decision how to transform the test cases. Once a pattern is selected, then it has to be configured, i.e., a set of method fragments has to be instantiated. Then, the instantiated fragments are customized regarding the functionality they are transforming. Customization of fragments means specifying them at a level of details so that it provides guidance during the enactment. The outcome, the *Context-Specific Migration Method Specification*, defines how to do the migration by defining the activities to be performed and the artifacts that should be generated.

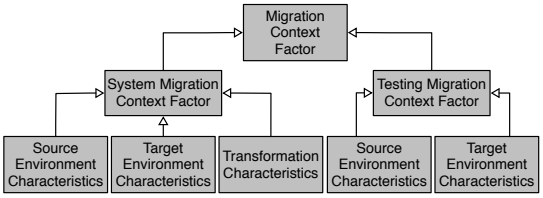

Figure 4: An Excerpt of the Context Model.

3.2 Method Enactment

During *Method Enactment* the context-specific tools are developed that are required for the automation of the migration method or part of it. Thereafter, the migration method is performed as defined in the migration method specification.

Having the migration method specified, every specified activity that shall either be performed automatically or semi-automatically, an appropriate tool is implemented. In general, there are two different types of tools: tools which are used in any migration method and specifically developed tools for a certain method. For example, if a given migration method defines a transformation which should be automated, then a model transformation engine that can execute model transformation rules is required. If, on the other hand, a migration method, i.e., a given method fragment requires a specific tool, e.g., a semantic analyzer, then it should be specifically developed. In general, the *Tool Implementation* activity requires a knowledge for model-driven engineering and development of reengineering tools. As a result of this activity, an integrated tool chain is delivered. During the last activity of the process, named as *Transformation*, the actual transformation of the original test cases occurs. This activity encompasses the enactment of the previously developed migration method and the corresponding tool chain. By enacting the migration method, the source test cases are being transformed, i.e., migrated to the new environment and can be used to validate the system migration.

4 CASE STUDY

Our method was applied in an industrial project where the main goal was to migrate parts of the well known Eclipse Modeling Framework (EMF) along with the Object Constraint Language (OCL) from Java to C# (Figure 5). As EMF and OCL are stable and well-tested and all test cases are available on public code repositories, our goal was to reuse the OCL test cases in order to validate the OCL functionality in the target environment. The main change implemented by the system migration, besides switching from Java to C#, was to change from a Just-In-Time (JIT) compilation in the old system to an Ahead-Of-Time (AOT) compilation of OCL constraints in the migrated system. The test cases which have similar structure

are selected for automated migration. For those test cases where it is difficult to identify some repeating structure, reimplementation is a more suitable solution. Additionally, an interlayer testing framework `TestOCL` exists in order to simplify the test cases. It provides domain-specific assert functions for the OCL domain, e.g., `assertQueryTrue`. Regarding the language, the source system is implemented in Java under Java Platform. Regarding the architecture, OCL is implemented in that a native OCL expressions are specified and using the JAVA API they are evaluated. The language in the target environment is C# under the .net platform. The OCL functionalities are implemented directly in the corresponding classes that are specified in the migrated system. Regarding the system transformation characteristics, the activities are ranging from manual to completely automatic. Those transformations that were performed automatically, have been formalized using transformation rules specified in Java. The process of transformation was supported by implementing transformation rules based on Xtend. At highest importance for the reflection of the changes to the test cases are the transformation rules specified for the transformation of OCL expression to C#. The source test environment is JUnit, with an additional interlayer extension, the *TestOCL*. The test cases are unit level test cases, separated in different test suites according the functionality they are addressing. The anatomy of the test cases in the target environment is that a single test case may contain multiple assertions, normally specified using the assert functions provided by the *TestOCL* extension. The target environment language, in this particular case C#, implies usage of *MSUnit*.

To automate the test case migration, two Eclipse plug-ins were developed to automate the reengineering activites. The overall result was an automated migration of over 92% of 4000 existing jUnit test cases. 8% of the test cases (mainly regression test) were not migrated automatically, due to the irregular structure which complicates an automation.

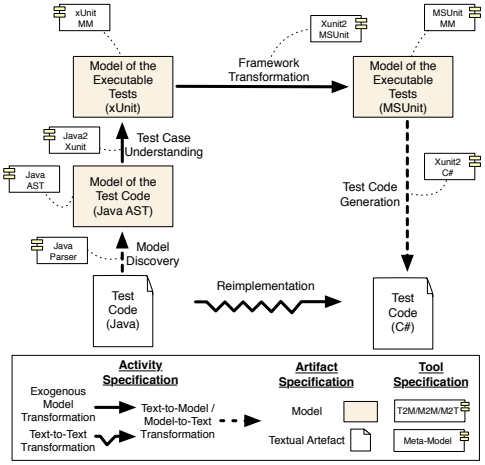

Figure 5: Excerpt of the Test Case Migration Method Specification.

5 RELATED WORK

There are already several approaches that implement the forward engineering process of test case reengineering activities. The benefit from the separation of PIMs and PSMs in the generation and execution of tests is presented in (Heckel and Lohmann, 2003). A model-driven testing methodology which transforms UML system models to test-specific UML Testing Profile models is proposed in (Dai and Dai, 2004). There are also appraoches that present reverse engineering activities for test cases. In (Hungar et al., 2003), test models are extracted out of test cases by means of automata learning. The benefit of reusing test cases has been detected in a lot of software migration projects. In the SOAMIG (Zillmann et al., 2011) migration project, existing test cases are used for the analysis of the legacy system. MoDisco (Bruneliere et al., 2010), on the other hand, is a generic and extensible framework which follows the principles of Model-Driven Reverse Engineering. However, migration of test cases is still not addressed by this framework. The ARTIST (Menychtas and Al., 2014) project is the most interesting project as they also advocate migration of the test cases in a model-driven way, i.e., in a similar way the system has been migrated, thus reusing, above all, the developed tools. Our work differs in that way, that we propose a modular construction of migration methods, by considering the migration context and also by addressing the co-evolution issue.

Regarding method engineering, depending on the degree of controlled flexibility, there are different categories of method engineering approaches, namely those that provides 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. The problem is however, that the process of changing of the provided method is not guided by a method engineering process. The following approaches follow this strategy: REMICS(Mohagheghi, 2010), SOAMIG (Zillmann et al., 2011), Reference Migration Process (ReMiP) (Sneed et al., 2010), and ARTIST (Menychtas and Al., 2014). 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. (Khadka et al., 2011) is a method engineering approach that enables modular construction, but is specialized for migration to service-oriented environments. The MEFiSTO Framework (Grieger et al., 2016) overcomes this issue by providing a general solution for modular construction of situation-specific migration methods. However, it lacks the support for migration of test cases, namely the consideration of the test context, as well as the analysis of the impact that the system changes have on the test cases.

6 CONCLUSION AND FUTURE WORK

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. The method engineering process provides a guidance on development and enactment of migration methods. We applied our framework in an industrial project in which a migration of part of EMF from Java to C# was performed. Using this approach, a model-driven test case migration method was developed and enacted and as a result, 4000 unit test cases were migrated from one environment to another, most of them completely automatic. As future work we plan to extend our framework by adding a post-migration phase, in which the validity of the test case migration is being assessed. This validation phase should check whether the functionality of the tests is preserved, i.e., whether the tests after migration are still testing the same part of the system functionality.

REFERENCES

Bisbal, J., Lawless, D., Bing Wu, B., and Grimson, J. (1999). Legacy information systems: issues and directions. *IEEE Software*.

Bruneliere, H., Cabot, J., Jouault, F., and Madiot, F. (2010). MoDisco. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*. ACM Press.

Chikofsky, E. J. and Cross, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*.

Dai, Z. R. and Dai, Z. R. (2004). Model-Driven Testing with UML 2.0. *Computing Laboratory, University of Kent*.

ETSI, E. T. S. I. (2016). ETSI ES 203 119–1: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics, v1.3.1.

Fuhr, A., Winter, A., Erdmenger, U., Horn, T., Kaiser, U., Riediger, V., and Teppe, W. (2012). 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.

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

Heckel, R. and Lohmann, M. (2003). Towards model-driven testing. In *Electronic Notes in Theoretical Computer Science*. Elsevier.

Hungar, H., Hungar, H., Margaria, T., and Steffen, B. (2003). Test-Based Model Generation for Legacy Systems. *IEEE International Test Conference*.

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

Jovanovikj, I., Güldali, B., and Grieger, M. (2016b). Towards applying model-based testing in test case migration. *Softwaretechnik-Trends, Proceedings of the 39th Workshop Test, Analyse und Verifikation (TAV)*.

Kazman, R., Woods, S., and Carriere, S. (1998). Requirements for integrating software architecture and reengineering models: CORUM II. In *Proceedings Fifth Working Conference on Reverse Engineering*. IEEE Comput. Soc.

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

Menychtas, A. and Al., E. (2014). Software modernization and cloudification using the ARTIST migration methodology and framework. *Scalable Computing: Practice and Experience*.

Mohagheghi, P. (2010). Reuse and Migration of Legacy Systems to Interoperable Cloud Services- The REMICS project. In *Mda4ServiceCloud*. Springer, Berlin, Heidelberg.

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

OMG (2013). UML Testing Profile (UTP), Version 1.2.

Sneed, H. M., Wolf, E., and Heilmann, H. (2010). *Software-Migration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung*. dpunkt Verlag.

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