An Approach for Creating KDM2PSM Transformation Engines in ADM Context: The RUTE-K2J Case

Guisella Angulo
UFSCar
São Carlos, Brazil
guisella.armijo@ufscar.br

Daniel San Martín
UFSCar
São Carlos, Brazil
daniel.santibanez@ufscar.br

Bruno Santos
UFSCar
São Carlos, Brazil
bruno.santos@dc.ufscar.br

Fabiano Cutigi Ferrari
UFSCar
São Carlos, Brazil
fabiano@dc.ufscar.br

Valter Vieira de Camargo
UFSCar
São Carlos, Brazil
valter@dc.ufscar.br

ABSTRACT

Architecture-Driven Modernization (ADM) is a type of software reenginering that employs standard metamodels along the process and deals with the whole system architecture. The main metamodel is the Knowledge-Discovery Metamodel (KDM), which is language, platform independent and it is able to represent several aspects of a software system. Although there is much research effort in the reverse engineering phase of ADM, little have been published around the forward engineering one; mainly on the generation of Platform-Specific Models (PSM) from KDM. This phase is essential as it belongs to the final part of the horseshoe cycle, completing the reengineering process. However, the lack of research and the absence of tooling support hinders the industrial adoption of ADM. Therefore, in this paper we propose an approach to support engineers in creating Transformation Engines (TE) from KDM to any other PSM. This approach was emerged from the experience in creating a TE called RUTE-K2J, which aims at generating Java Model from KDM. The transformation rules of RUTE-K2J were tested considering sets of common code structures that normally appears when modernizing systems. The test cases have shown the transformation rules were able to generate correctly 92% of the source code that was submitted to the transformation.

CCS CONCEPTS

• **Software and its engineering → Software post-development issues**;

KEYWORDS

Model transformation, KDM, PSM, Java Model

ACM Reference Format:
Guisella Angulo, Daniel San Martín, Bruno Santos, Fabiano Cutigi Ferrari, and Valter Vieira de Camargo. 2018. An Approach for Creating KDM2PSM Transformation Engines in ADM Context: The RUTE-K2J Case. In *XII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '18), September 17–21, 2018, Sao Carlos, Brazil,* 10 pages. https://doi.org/10.1145/3267183.3267193

SBCARS '18, September 17–21, 2018, Sao Carlos, Brazil

ACM ISBN 978-1-4503-6554-3/18/09…$15.00
https://doi.org/10.1145/3267183.3267193

1 INTRODUCTION

Software systems are usually acknowledged as *legacy systems* when they exhibit two main characteristics: *(i)* they have high maintenance costs (effort/time/resources); and *(ii)* they are still essential to support current business processes. Clearly, these systems cannot be discarded since they retain valuable business knowledge that was incorporated along years of maintenance [6]. For many years, software reengineering has been sold as a solution to this problem, since it retains all the knowledge of these systems. However, a study [22] has shown that more than 50% of the reengineering projects fail, and one of the main reasons is the lack of standardization, which hinders the reusability of solutions and interoperability among reengineering/modernization tools [12].

In 2003, the Object Management Group (OMG) took its first steps towards the proposal of Architecture-Driven Modernization (ADM). The main idea was to define standards for the reengineering process in order to promote industry consensus on the modernization solutions and elevate the success in modernization projects. Many companies have demonstrated interest in ADM philosophy. In the ADM Vendor Directory Listing of the ADM website [4], there are around 30 IT companies listed as OMG partners that have some kind of interested in the ADM philosophy. Some companies are specialized in modernizing systems, while others offer this type of service in their solutions portfolio.

The Knowledge Discovery Metamodel (KDM) is the main ADM metamodel and its goal is to represent/capture all the system architecture independently from platform and language, so it is a PIM (Platform Independent Model). KDM is expected to be applied along all modernization phases. More specifically, KDM can be applied (*i*) in *reverse engineering*, in which the legacy system is parsed and a KDM instance that represents the legacy system is obtained (another task in this phase is the identification of problems); (*ii*) in the *restructuring phase*, in which the KDM instance is restructured/refactored to solve the identified problems, and a new, modernized KDM instance is generated; and (*iii*) in the *forward engineering*, in which the modernized KDM is used as input for re-generating the system, completing the modernization cycle. Within this last phase, there are still two steps: the generation of a PSM from the KDM, and the generation of the source code from the PSM.

In the literature it is possible to find approaches that focus on the reverse engineering phase using KDM, such as MoDisco [7],

Gra2Mol [9] and Three-Phase Approach [28]. However, little research has been conducted and published about forward engineering with KDM. There are some research on the entire modernization process, so they have addressed the forward engineering phase somehow. However, the focus of these initiatives was not on bringing contributions to the forward engineering phase, thus they do not provide enough details about it [20, 24].

Therefore, in order to fill this research gap, we have developed an approach for supporting software engineers in the creation of transformation engines (TE) that transform KDM to any other PSM. The approach has three phases and is iterative and incremental, where in each cycle, one transformation rule is developed/evolved. This approach emerged from the experience in creating a TE called RUTE-K2J that takes a KDM instance (possibly a modernized one) as input and automatically generates a Java model from it. Java model is a well-known PSM (Platform Specific Model) and there are code generators that take it as input [1, 3]. An important point here is that RUTE-K2J contributes to the second-to-last step of ADM horseshoe model (step *kdm2psm* in Figure 1), opening many research possibilities for researchers and also companies testing the promises of ADM regarding reusability, effectiveness, etc.

RUTE-K2J was evaluated with the execution of test cases with the aim of guaranteeing certain level of correctness when performing the transformations. During the execution of the test cases, we used the support of Modisco tool to generate a KDM instance (input of the RUTE-K2J) and the Acceleo tool for generating source code from the Java Model (output of the RUTE-K2J). The source code generated by Acceleo allowed us to compare the produced code with the original code to evaluate the correctness. Our evaluation showed that 92% of the source code that was submitted to the transformation was generated correctly.

The remainder of this paper is organized as follows: Section 2 presents necessary background related to ADM and KDM, and model transformations. Then, in Section 3 presents the guidelines for creating KDM2PSM transformation engines. In Section 4, we present the RUTE-K2J Transformation Engine. In Section 5, we present the validation of RUTE-K2J. In Section 6, we summarize related work and, finally, in Section 7 we draw some conclusions and describe plans for future work.

2 BACKGROUND

2.1 ADM & KDM

In 2003, OMG proposed the Architecture-Driven Modernization (ADM) [9, 12] initiative which follows the MDA principles. ADM promotes system modernization based on the use of models at different abstraction levels [26]: CIM, PIM and PSM levels. Furthermore, it proposes the use of automated transformations to generate new systems from legacy systems by following a horseshoe process that is shown in Figure 1.

The ADM modernization cycle has three phases (i): reverse engineering, (ii) restructuring and (iii) forward engineering. In the reverse reengineering phase, the knowledge is extracted from source code and a PSM is generated. The PSM model serves as basis for generating a PIM called KDM. Then this PIM can serve as basis for creating a CIM. In reestructuring phase, refactorings are performed in the legacy instance in order to get an improved version of the

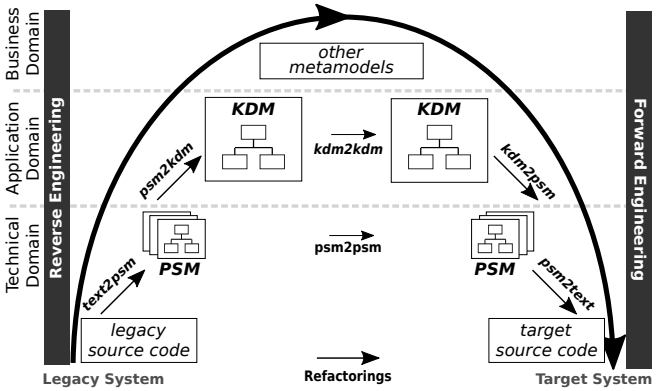

Figure 1: Horseshoe modernization model

system. Finally, in the forward engineering, the refactored instance is transformed in source code again.

ADM defines seven standard metamodels, but currently only three of them are available: Abstract Syntax Tree Metamodel (ASTM), KDM and Software Metrics Metamodel (SMM) [12].

KDM is an OMG metamodel adopted as ISO/IEC 19506 capable of representing a complete software system. It can be seen as a family of metamodels that share the same vocabulary and terminology, facilitating the relationships among metaclasses in different abstraction levels. The specification is organized in four layers: Infrastructure Layer, Program Elements Layer, Runtime Resource Layer, and Abstractions Layer where each layer is based on the previous one. These layers are further organized into packages and each one corresponds to a certain independent facet of knowledge about the software, such as the Code View, Structure View, Data View among others. Current tools that generate KDM instances consider the*Program Elements Layer*, which contains the following packages: The Code package defines metamodel elements that represent low-level building blocks of software, such as procedures, data types, classes, variables, and so on. Among the elements of the package are: ClassUnit, MethodUnit, StorableUnit, InterfaceUnit, PrimitiveType, among others; the Action package defines metamodel elements that represent statements as the relationship endpoints and most low-level KDM relationships.

KDM is designed to enable knowledge-based integration between tools. More specifically, KDM uses Meta-Object Facility (MOF) to define an interchange format between tools that work with existing software as well as an abstract interface (API) for the next-generation assurance and modernization tools. Therefore, one can create approaches and tools to: (i) apply refactorings [10], (ii) perform architecture conformance checking [14], (iii) mine crosscutting concerns [21], etc.

2.2 Model Transformations

Model(-to-model) transformation is a core asset in model-driven engineering (MDE), where models are first-class entities. It is the process of converting models into other models for the purposes of supporting rigorous model evolution, verification, refinement, and code generation [11]. Given a source metamodel MM_A, a source model M_A (as an instance of MM_A) and a target metamodel MM_B, define and execute a transformation T_{AB} which generates a model

M_B that is an instance of MM_B and suitably corresponds to M_A. The transformation T_{AB} conforms to its metamodel MM_T, but involves also the metamodels MM_A and MM_B to define the transformation rules. All the metamodels finally conform to the meta-metamodel *MOF* at model layer *M*3. This way, a transformation can have multiple source models, and produces multiple target models.

Model transformation can be classified from different perspectives. According to the metamodels, to which the source and target models conform to, two kinds of transformation can be distinguished: *exogenous* and *endogenous*. An endogenous transformation, also called an *inplace* transformation, translates a source model into a target model that conforms to the source model's metamodel, e.g., a refactoring of a KDM instance. In contrast, an exogenous transformation, or *out-place* transformation, uses different source and target metamodels, e.g. transforming a KDM instance to JAVA instance model [15]. According to the transformation directions, a transformation can be *unidirectional* or *bidirectional*. An unidirectional model transformation has only one execution direction, that is it always modifies the target model according to the source model. In case of bidirectional transformation, the source model may be changed along with the target model if the transformation is executed in the direction of target-of-source.

3 APPROACH FOR CREATING KDM2PSM TRANSFORMATION ENGINES

This section presents our approach for creating transformation engines from KDM to PSMs. The main goal of the approach is to support the elaboration of mappings between the KDM and a particular PSM by comparing instances of these metamodels. The result is used to develop the transformation rules that make up the transformation engine.

The approach consists in one activity called *Choosing PSM metamodel and reverse-engineering Tool* and three phases, as depicted in Figure 2. The goal of the activity is twofold; the first one is to define which PSM (Java Model, C# model, Service-Oriented Model, etc) will be used as the target metamodel because the forward transformation will generate instances of this PSM. The second one is to choose a reverse-engineering tool/parser that generates a PSM from source code. The output of this activity are the PSM and the tool for reverse-engineering.

The *Phase I* called *Preparing Initial Artifacts* has three activities with the purpose of generating the PSM and KDM instance artifacts used in the next phases. The *Phase II* called *Developing KDM2PSM transformation engine* has three activities with the purpose of developing the transformation rules that conform the transformation engine. Finally, the *Phase III* called *Validating Transformation Rule* has two activities with the purpose of testing the transformation rule and verify the completeness of the resulting PSM Model.

Our process is iterative and incremental because in each phase the activities could be repeated several times until there is no language statement to be created or the software engineer stop the process. Note that every time a cycle is completed, a transformation rule is created.

The activities in each phase are shown in Figure 3 and described below.

Phase I: Preparing Initial Artifacts.

This first phase aims to generate the artifacts used in the next phases.

The first activity: Defining the Code Snippet to be Represented by the Models. The goal is to choose the source code structure that a rule will be written for. For example, if an engineer choose the source code element *while* then the resulting transformation rule of this iteration will recognize the representation of the *while* in KDM and will generate it in the target PSM. This activity depends on the purpose of the code generation. Sometimes it will be necessary to generate complete source codes that take into account method bodies, types and relationships. In other cases, it may be only necessary to generate just method signatures, classes, attributes and parameters. After that, the modernization engineer must implement or obtain a simple piece of code of the structure. This will serve as an input for the next activities. The output of this activity is a piece of source code implemented or obtained by the modernization engineer.

The second activity: Generating the Sample PSM. The goal is the generation of the PSM instance from the piece of source code of the previous activity. The PSM is an important artifact in the overall process because two reasons. Firstly, in the activity 4 it helps in the identification of the equivalent metaclasses between both metamodels. Secondly, in the activity 7 it is used as an oracle to verify the completeness of the PSM output. Thus, the modernization engineer uses the Code2PSM tool and provides as an input the piece of source code. The result of this activity is the PSM instance, called from now as the *Oracle PSM*.

The third activity: Generating the KDM instance. The goal is the generation of the KDM instance. This instance assists the software engineer in the identification of the metaclasses that are equivalent between the metamodels and it provides the input elements for testing the rule in the activity 6. To create the artifact, the modernization engineer uses a modernization tool that generates the KDM instance (PIM) from source code. In the literature, there are some tools that can automate this process such as: Bruneliere et al [7] and the tool MoDisco for generates KDM instance from Java source code; Feliu Trias et al [23], they provide a tool for obtain the KDM instance from PHP source code; Christian Wulf et al [28], they present an approach to transform C# programs to KDM and the commercial software BLUAGE [5] can transform Cobol code to KDM.

Phase II: Developing KDM2PSM transformation engine. It is the main phase and its goal is the development of the rule responsible for transforming the KDM instance that represents the chosen source code structure in a PSM instance representing the same structure.

The fourth activity: Creating mapping KDM - PSM. This activity has as goal to establish the mapping between the metaclasses of the metamodels. To this end, the modernization engineer has to perform a comparative analysis between the KDM instance, output of the activity 3, and the Sample PSM, output of the activity 2. This comparison between the models (XMI files) helps to identify the metaclasses and its attributes used for represent the structure chosen. For example Figure 4, shown the comparison between the KDM instance (A) with the PSM model (B) for the structure *while*. The equivalence elements are denoted with the numbers (1), (2) and (3). The result of this activity is the artifact called *the KDM-PSM*

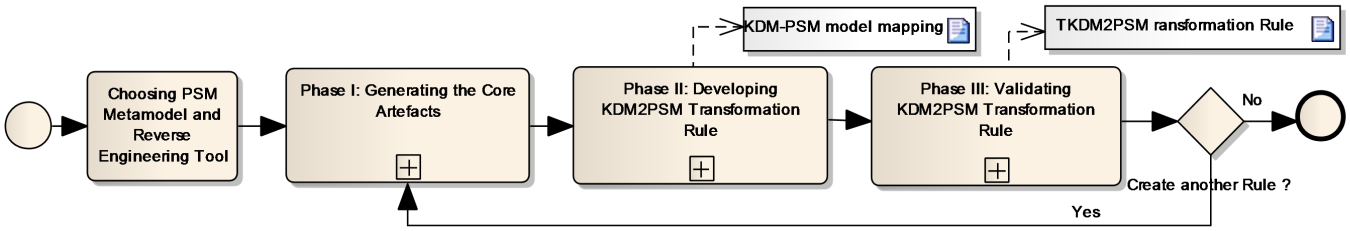

Figure 2: Phases of the Approach for Creating KDM2PSM Transformation Engine

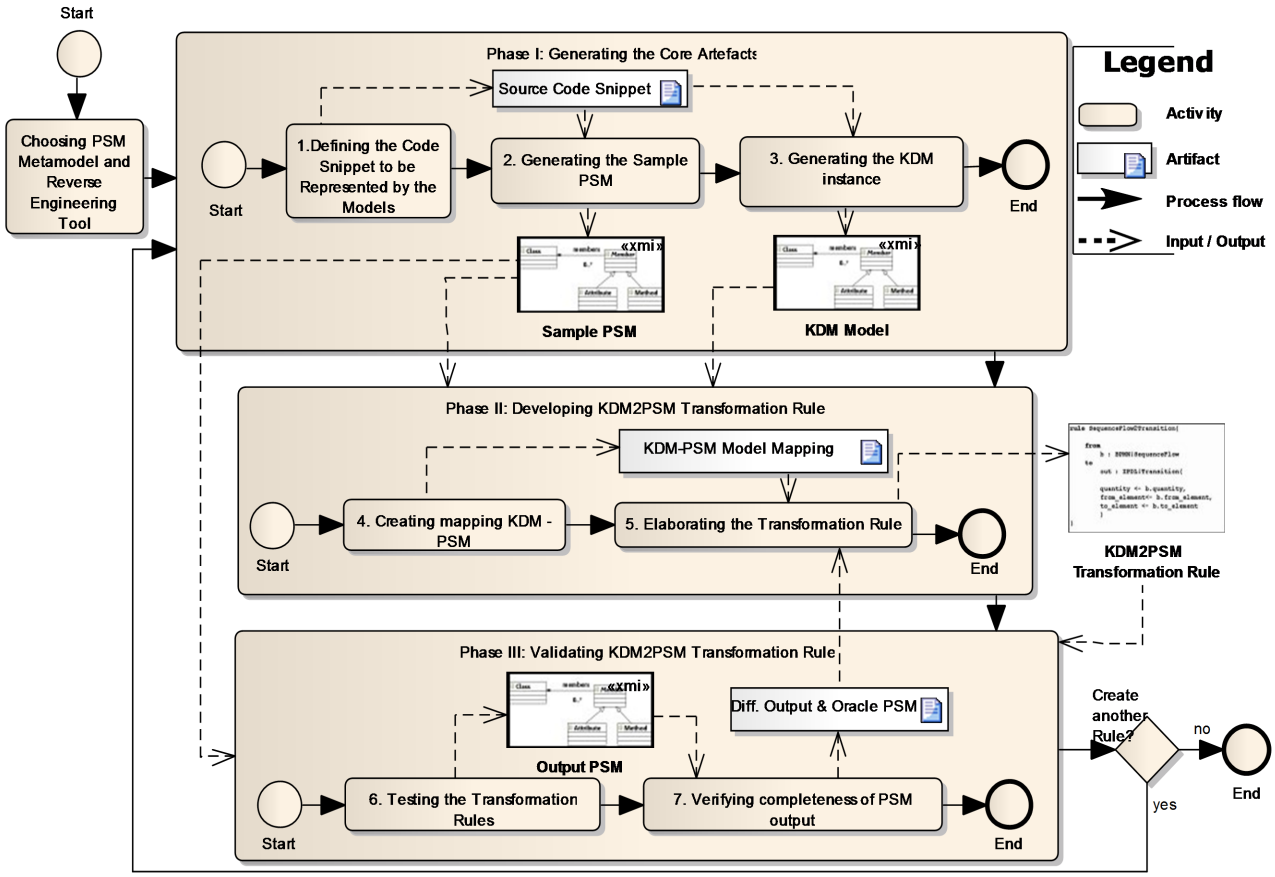

Figure 3: Phases and Activities of the Approach for Creating KDM2PSM Transformation Engine

model mapping, which record the equivalence between PSM-to-KDM elements. This artifact is actively consulting and updating in each iteration.

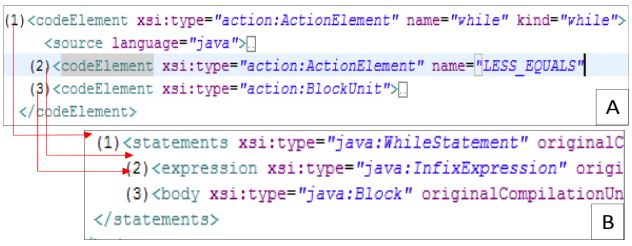

Figure 4: Comparison between KDM and PSM model

The fifth activity: Elaborating the transformation rule. The goal of this activity is to develop the transformation rule to transform from KDM instance to PSM instance, preserving the chosen source code structure embodied in the source model. Firstly, the modernization engineer using the *the KDM-Java model mapping*, the metamodel documentation and the transformation tool documentation has to establish the source KDM metaclass and the target PSM metaclass of the transformation, including filters or conditions for delimit the source. Secondly, develop the rule body, he must place on the left side each attribute of the PSM metaclass assigning the correspondence KDM metaclass attribute, on the right side, according to the mapping in the *the KDM-PSM model mapping*. The modernization engineer must know the syntax of the transformation language to perform this activity. Finally, one or many functions must be implemented to complete the information that

cannot be obtain directly from the source KDM instance. Due to the low-level abstraction of the PSM model, it needs more specific information than is provided by the KDM instance.

Phase III: Validating Transformation Rule. This phase has as goal testing the developed transformation rule and verify the completeness of the PSM output.

The sixth activity: Testing the Transformation Rule. The goal is to test the transformation rule developed to verify that they are correct. In order to achieve this goal, the modernization engineer has to execute the transformation rule using the transformation tool kit and configure the KDM instance as input. The output of this activity is the PSM instance called *Output PSM*.

The seventh activity: Verifying completeness of PSM output. The goal of this activity is to verify the completeness of the output PSM generated in the last step, when comparing with the sample PSM that was generated in the activity 2. In order to achieve this goal, the modernization engineer has to compare the both model looking for differences in the structure; it is a manual work that can be facilitated with some free tool for compares XMI files. The output of this activity is a list of differences between the models. When differences are found, they must be corrected by returning to activity 6.

4 RUTE-K2J: THE TRANSFORMATION ENGINE

In the literature, several reengineering approaches describe how to transform a KDM instance into a particular technology using specific components and mapping. For example, from KDM-Extend to SQL-DML [17], from KDM to KD model [25] or from KDM to Metrics Model [8]. These works transform KDM instances in other metamodel instances in conformance with proprietary metamodels, hindering the reuse in other similar projects. Moreover, few works complete all the modernization process using the KDM instance in the forward transformation stage. For example, from KDM to ASTM[16], from KDM to Android Model [18] or from KDM to RIA-extended MDWE [20]. These works not make available the developed transformation rules for analysis, reuse or improvement of them.

There are a gap to transform KDM instances to other available and knowing model to complete the forward engineering stage in ADM Modernization Process. To close this gap, we propose the *Rule-based Transformation Engine KDM2JAVA* (RUTE-K2J) that has as mainly objective to facilitate the transformation from KDM instance to Java Model and use it on modernization use cases. The RUTE-K2J benefits are the following:

- Facilitating the exogenous transformation from KDM instance to Java model. It is valuable to note, the input KDM instance generated only supports the *Code* and part of the *Action* package;
- Allowing the use in modernization use cases. RUTE-K2J collaborates with the Forward Engineering stage of the ADM horseshoe model;
- Allowing the use of the Java Model J2SE5 (output of the transformation) that is a reflection of the Java language;
- Using the Java metamodel that is defined in the widely used metalanguage *Ecore*, which allows the modification, extension or reuse by metamodeling tools. This metamodel is available in the source of plug-in *org.eclipse.gmt.modisco.java*;
- Allowing the analyses, reuses and improvements because is a Open source project.

RUTE-K2J was developed using the following technologies: *i*) the ATL transformation language [2]; *ii*) the OCL (Object Restriction Language) norm for data types and declarative expressions; *iii*) Eclipse Modeling Framework (EMF); *iv*) Modisco and the discoverers that allow the generation of the Java Model and KDM from the source code; *v*) Discover-Advanse, which includes improvements made in the ATL rules to generate a refined KDM instance.

This first version of the *RUTE-K2J* tool is composed for 55 transformation rules, 28 Helpers and 10 Lazy rules. The Rute artifacts are described below.

i) The *KDM-PSM model mapping*. This artifact shows the mapping between the equivalent metaclasses of KDM and Java model. The mapping guide the developing of the forward transformation rules because identify the input KDM metaclass for generate the correct output Java metaclass instance. Table 1 shows part of *KDM-PSM model mapping* artifact, the first and second column refers to the KDM metaclass and the condition used as filter, the third column refers to the Java classes. Observing the table, we can realize that the *Action Element* KDM metaclass can be transformed into several Java Metaclasses (*IfStatement*, *InfixExpression*, *SwitchStatement*, etc.), for this reason, we must use as a condition to limit the input source the *Kind* attribute of KDM metaclass. This artifact is important because the KDM has many elements where the distinction is in the attribute called *kind*, with no value in some enumeration of elements. The complete artifact can be access in the following URL: https://goo.gl/xnfS6p.

Table 1: Partial KDM2Java Model Mappings

KDM Instance		JAVA Model
Metaclass	**Filter**	**Metaclass**
CodeModel CodeModel	name='Nome_projeto' name='External'	Model
Package	-	Package
ClassUnit	name <> 'Anonymous type' name = 'Anonymous type'	ClassDeclaration AnonymousClassDeclaration
MethodUnit	kind = constructor kind = method	ConstructorDeclaration MethodDeclaration
StorableUnit	kind <> local kind=local	FieldDeclaration VariableDeclarationFragment
BlockUnit	-	Block
ActionElement	kind = 'if' kind ='infix expression' kind='postfix expression' kind='switch' kind='while' kind='method invocation' kind='class instance creation' kind='return'	IfStatement InfixExpression PostfixExpression SwitchStatement WhileStatement MethodInvocation ClassInstanceCreation ReturnStatement

ii) The *KDM2JAVA Transformation Rules inventory* artifact shows the inventory of the developed transformation rules ATL. The transformation rules compose the core of the RUTE-K2J tool and allow the transformation of KDM instances to Java Models preserving the information during the process. Table 2 shows part of the *KDM2JAVA Transformation Rules inventory* artifact. We can observe that the name of the rule evidence the metaclass of origin and destination in the transformation. For example, the rule *CodeModelToJavaModel* has as objective transform the KDM metaclass

Table 2: Partial Transformation Rules

Nº	Rule name and its purpose
1	**CodeModelToJavaModel:** Main rule for structuring the Java Model from the KDM instance
2	**PackageToJavaPackage:** Transform the *Package* metaclass of the KDM to *Package* metaclasses of the Java metamodel
3	**ClassUnitToClassDeclaration:** Transform the *ClassUni* metaclass of the KDM to the *ClassDeclaration* metaclass of the Java metamodel
4	**ActionElementToInfixExpression:** Transform the *ActionElement* metaclass of the *infix expression* type of the KDM instance to the *InfixExpression* metaclass of the Java Model
5	**ActionElementToIfStatement:** Transform the *ActionElement* metaclass of the *prefix expression* type of the KDM instance to the *PrefixExpression* metaclass of the Java Model
6	**ActionElementToForStatement:** Transform the *ActionElement* metaclass of type *for* from the KDM instance to the *ForStatement* metaclass of the Java Model

CodeModel to the Java model metaclass *Model*. This rule is the principal for structuring the model and articulate the other rules. The complete artifact with 55 *Transformation Rules inventory* can be access in the following URL: https://goo.gl/CZQXM5.

Table 3: Partial Helpers

Nº	Helper Name and purpose
1	getOrphanTypes: Helper that returns a sequence of *datatypes* elements
2	getCompilationUnit: Helper that returns the inventory of the physical artifacts of the system
3	getParametersMethod: Helper to get the parameter sequence of the Method
4	getReturns: Helper to get the sequence of return elements of the Method
5	checkElementoExternal: Helper recursive to verify if the element belongs to the *External Model* in the KDM instance in order to put the attribute 'proxy = true' in the Java Model

Table 4: Partial Lazy Rules

Nº	Lazy Rule Name and Purpose
1	setOrphanTypes: Defines the attributes of the orphanTypes metaclass of the Java Model
2	setCompilationUnit: Defines the attributes of the CompilationUnit metaclass of the Java Model
3	SetParametros: Defines the parameters attributes of the method in the Java Model
4	SetTypeParametrosRetorno: Defines the data type of the return parameter of the Method

iii) The *ATL Helper Inventory* artifact. In the ATL context, the helpers can be viewed as the equivalent to methods. They can be called from different points of an ATL transformation by rules or by other Helpers making possible define factorized ATL code, calculate information, etc. Table 3 shows part of the developed Helpers. Observing the table, we can realize the Helper *getOrphanTypes* helps us to obtain a sequence of *datatypes* elements presents in the source Model. The complete artifact with 28 *ATL Helper Inventory* can be access in the following URL: https://goo.gl/HaCYYb.

iv) The *Lazy rules inventory* artifact. In our project, the Lazy rules is used in the characterization of collections elements, it must be explicitly invoked by the transformation rule. Table 4 show part of the lazy rules. Observing the table, we can realize the lazy rule *setOrphanTypes* helps us assign values for each attribute returned by the *getOrphanTypes* Helper. The complete artifact with 10 *Lazy Rules Inventory* can be access in the following URL: https://goo.gl/Y45cff.

4.1 Problems Faced

During the process of construction of the *RUTE-K2J* tool, i. e., the establishment of the mapping between the equivalent elements of the metamodels and the development of the transformation rules, we face several difficulties. The main ones are:

Abstraction level: The target Java model due to the lower level of abstraction has metaclasses that need more information than the one provided by KDM instances. For example, the *CompilationUnit* (Java metaclass) has the atributes: *name*, *originalFilePath*, *types* and *imports*. On other hand, the metaclass equivalent the *InventoryItem* (KDM metaclass) only has the attribute *name* and *originalFilePath*. The missing information was obtained developing *Helpers ATL* when was possible.

Order of elements: KDM has metaclasses with attributes that do not have an explicit assignment order. For example, the *ActionElement* metaclass has many *codeElements* attributes with no label indicating the assignment order, we deduce that this is according of the order of appearance. While in the Java model, the metaclasses present attributes with names that indicates the right assignment order. For example, the *RightOperand* and *leftOperand* attributes of the *InfixExpression* metaclass.

External Model: In a KDM instance has a model called *External* where stores the references of the external classes (imported classes). However, in the method body, expressions such declaration of local variables with initial values assignment have these values placed in this model. The combination of these elements in the external model difficult: (i) Retrieving the assigned value and (ii) recognizing and transforming each element in the *External* Model.

5 VALIDATION SETUP

We have conducted an evaluation of the Transformation Engine whose goal was to guarantee certain level of correctness when performing the transformations. More specifically, we intended to verify if the developed rules are correct and if the combination of them, in an aleatory way, result in correct transformations. The main goal is raising evidences of quality in the transformation rules so modernization engineers can use it in their projects. In order to conduct the evaluation, we elaborated a testing strategy for executing each of the 55 transformation rules at least one time.

5.1 Scoping

Defining the scope of an experiment comes down to setting its goals. We used the organization proposed by the Goal/Question/Metric (GQM) [27] template to do so. According to this goal definition template,the scope of our study can be summarized as follows.

Analyze whether all the RUTE transformation rules works as expected
for the purpose of evaluation
with respect to correctness of all transformations rules[1]
from the point of view of the researcher
in the context of software engineer, i.e., engineer transforming KDM instance to Java Model.

[1]Herein, a transformation rule is correct when it successfully transform a source element into an expected (oracle) element

5.2 Sample Selection

We focused our analysis on open source projects so that our study could be easily replicated and extended. Given that our sample was randomly selected from open sources "repositories", we tried to include a wide range of source code that differ in size and complexity. Therefore, we have selected and downloaded some Java repositories ordered by popularity in GitHub.

We adhered to the following criteria in constructing our sample:

- *Open-source projects:* we randomly selected open-source program hosted in GitHub repositories. We followed the guidelines proposed by Kalliamvakou et al. [13] during the construction of our sample.
- *Java:* has at least 90% of the code repository effectively written in Java.

5.3 Operation

In this section we describe the experiment operation, which was divided into two parts: preparation and execution.

Preparation: RUTE tool is composed of 52 standard and 3 abstract transformation rules. Each rule was individually developed with a specific objective, i.e., to generate control structures *If*, *For*, creation of methods, attributes of the Classes, initialization of variables, etc. However, these rules show dependence on each other to generate a complete Java source code. In this context, we developed a strategy to validate the fulfillment of the transformation rule objective and execute each transformation rule at least once in the test cases. The KDM instance generated from random and complete Java program comprises the test case. We define the following steps for elaborate the test cases.

- Creating the Rule Matrix: Elaborating a matrix with the transformation rules, identifying as follows: first, the identification of the main rules must be marked with "*". The main rules are responsible for the generation of the basic structure of Java Model. Second, the optional dependence on other rules that must be marked with "**". Notice that we call this as *optional dependence on rules* because there are many possible combinations between the rules for generate structures. We only marked dependence based on the examples used in the process of creating the transformation rules. Finally, the indirect dependence on other rules must be market with *ID-(Rule)*;
- Identifying *Leaf* Rules: In the Rule Matrix, we identify the rules that are not required by other rules to complete their execution. This means that no other rules depend on them. However, these rules may depend on several others. In order to manage that, we count the number of marks in each columns and select those with the value ''0". If we compare with a graph structure, the rule would be the leaf;
- Prioritization *Leaf* Rules: For each identified *Leaf* Rule, we count the number of marks in each row, so that result represents the number of dependencies on the other rules. After that, we make a prioritization considering the highest to lowest number of dependencies. We adopted as a criterion to elaborate the test cases considering *leaf* rules and their prioritization because each of them will guarantee the execution of the greatest possible number of rules;
- Test case: We search Java source code available on the web for covering the prioritized *leaf* rules. We opted for combining them in order to generate more complete java programs. Even so, it will not always be possible for a single program to cover all the leaf rules, i.e., all the rules which the *leaf* rule depends on. However, we must guarantee that this uncovered rule is executed at least once in some other leaf.

Execution: After identify the strategy for evaluate RUTE, the concrete actions are the following:

- Creating the Rule Matrix: To elaborate the matrix, we considered the 52 standard transformation rules, which are placed horizontally and vertically. Then, taking the first rule in the vertical are identified the rules in the horizontal of which it dependence on. We iterate with the following rules in the vertical until completing the 52 rules. Table 5 shows part of the *Rule Matrix*, we can observe that rule R30 dependence on R01 (JavaModel), R05 (JavaPackage), R11 (ClassDeclaration), R15 (MethodDeclaration) and R16 (BlockUnit) marked with "*", rules responsible for the creation of the Java model structure. Moreover, R30 optionally dependence on R14 (Constructor), R18 (ReturnStatement) or R21 (SwitchStatement) marked with "**", i.e., it depends of the type of structure that will generate. Finally, R30 indirectly dependence on R22 (BreakStatement) and R23 (SwitchCase) marked with *ID-R21*. This means, if R30 dependence on R21 it would indirectly depend on the execution of R22 and R23. R21, R22 and R23 together compose the *SwithStatement* structure.
- Identifying *Leaf* Rules: We count the occurrences in the matrix columns and identify the rules with result value equal to "0". This value indicates that no other rule depends on the rule in analyze. In Table 5 we realize that the R13 (FieldDeclaration) is a *Leaf* Rule.
- Prioritization *Leaf* Rules: We reduce the matrix considering as columns the identified *Leaf* rules. Then, we count the occurrences in the rows, ordering them from highest to lowest. Higher values tell us that this leaf rule can execute a greater number of rules, helping us to exercise a large number of transformation rules with few testing scenarios. Table 6 shows part of the *Leaf* Rules, we can observe the total number of occurrences for each rule. The R47 has 25 occurrences, this means that the programs where we use this rule could exercise up 25 rules of the set of 52.
- Test case: With the of the *prioritization*, we can elaborate and execute each test cases:
  Test Case elaboration: We combine leaf rules for each test case to use complete Java programs. This programs are chosen trying to cover all the *Leaf* rules and the rules that these depends on. In Table 7, we can observe the combination of *Leaf* rule for each test case. For example, the Test Case I exercises the *Leaf* rules R47 (10 Rules), R30 (7 rules) and R51 (2 rules) covering 19 rules (36.54%) of the set of 52. The Test Case II covers more eight new rules not executed before.
  Test Case execution: With the Java program chosen and with the help of the tool *Discover-Advanse*, we generated the KDM instance for each program. After that, we execute the tool RUTE-K2J configuring as input each KDM instance and obtaining as output the Java Model (XMI File). Next, with the aim to verify the completeness of the model, we use the Acceleo Plug-in to

Table 5: Rules Matrix

Rule	R01	R05	R11	R13	R14	R15	R16	R18	R20	R21	R22	R23	...
R13 TrasformStorableUnitToFieldDeclaration	*	*	*		-	-	-	-	-	-	-	-	...
R29 TransformWritesToSingleVariableAccess	*	*	*	-	**	*	*	-	-	-	-	-	...
R30 TransformReadsToSingleVariableAccess	*	*	*	-	**	*	*	**	-	**	ID-R21	ID-R21	...
...	...	...	...	...	....	...	...	...	...	...	...	...	...
Occurrences	51	47	42	0	34	38	38	5	4	8	7	7	...

Table 6: *Leaf* Rule Prioritization

Rule	R01	R05	R11	R13	R14	R15	R16	R18	R20	R21	R22	R23	...	Occurrences
R47 TransformValueToNumberLiteral	*	*	*		-	-	-	-	-	-	-	-	...	25
R48 TransformValueToStringLiteral	*	*	*	-	**	*	*	-	-	-	-	-	...	22
R50 TransformValueToBooleanLiteral	*	*	*	-	**	*	*	**	-	**	ID-R21	ID-R21	...	22
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Table 7: Test Cases

Test Case	R47	R30	R48	R50	R49	R31	R51	R52	R46	R10	R13	Coverage(1)	P% (2)
I	10	7					2					19	36.54
II	3					5						8	15.38
III		2	3	2								7	13.46
IV	2							2	1			5	9.62
V										2	2	4	7.69
VI						5						5	9.62
VII		3			1							4	7.69

generate Java source code from the Output Java model. Finally, the free tool *Pretty Diff* helps us to compare the original Java code and the Java Code generate by *Acceleo*. The result of this comparison is the different lines among source codes.

5.4 Results

Table 8 shows the seven test cases. Note that the first column shows the number of the Test case. The second and third column show the name of the source code project and the number of lines of code. The fourth column indicates the number of lines of code with differences. The fifth and sixth columns show the line of code with difference and the original. Finally, the seventh column details the difference between the fifth and sixth columns.

As result of the test cases execution, considering the 217 lines of code of the all programs chosen: 201 lines were generated correctly and 16 lines presented differences, i.e., 92 % of the code was generated successfully. The lines of code generated with differences are analyzed as follows:

- For all test cases, the reserved word *Static* is not being generated. Keeping track of the data preservation during the transformations, we find that this data is not brought by the KDM input model generated.
- Test Case I. Another difference was the order of the elements in the conditional part of the *if* structure. This is one of the main problems faced, the elements (instructions, assignments, etc.) in the structure (if, for, While, etc.) do not have a concrete specification in the order of apparition in KDM instance.
- Test case III. The *System.in* parameter for the creation of the *Scaner* class is absent. The lack of information in KDM instance was detected during the process of elaboration of the transformation rules.
- Test Case V. The resulting code is showing, after the creation of an anonymous class, an unnecessary parenthesis *"()"*. This is an error in the transformation rule when working together with the other rules. This error must be revised and corrected.
- Test case VI. In the creation of a structure *Array[]* the element *"[]"* is absent. The same reason of the cases before, the input model KDM do not have this information.
- Test case VII. It is missing a transformation rule for the comments, it will be developed for the next version of the tool. In addition, in the multiplication statement *(2 * b)* the order of the elements is incorrect, it is the same problem of the *Test Case I*. Moreover, it presents a lack of separation in the expression *"+++ i"*, here the problem is in the tool Acceleo and not in the transformation rules

As result of the evaluation, 92% of the code was successfully generated. This shows that the transformation rules, despite the limitations in the input model KDM, generate a Java model that preserve the information embodied in the source code.

5.5 Threats to Validity

The small number of source code examples we collected. However, when we are looking for source code samples, we tried to get representative samples, that is, those that presented common and usual source code structures;

All the combinations among rules were not exercised. Although we have exercised the rules in many combinations in each test case, many were left over. This is a threat we intend to solve in a future work.

The Discover-Advanse tool used for the generation of the KDM instance. Although the tool includes the improvements developed, it still does not generate a complete KDM instance.

6 RELATED WORKS

Most related works shown here are concentrate on showing the entire modernization process, evidencing some parts of the process. Next, we present some works that present transformations from KDM to other metamodels. However, several details of the transformations or the process they have used for elaborating the rules are not clear or shown in the paper.

Table 8: Rute-K2J Evaluation

Nº	File name	LOC	LOC-D	Original Source Code	Acceleo Source Code	Observation

I	ControlFlowStatements.java	61	4	public *static* int getMonthNumber
				public *static* void main(…
				if (month == null)
				if (returnedMonthNumber == 0)
					public int getMonthNumber
					public void main(…
					if (null == month)
					if (0 == returnedMonthNumber)
						Reserved word *Static* is absent
						Reserved word *Static* is absent
						Wrong order of the elements
						Wrong order of the elements

II	TestArray.java	20	1	public *static* void main(…	public void main(…	Reserved word *Static* is absent
III	GetAge.java	21	2	public *static* void main(… sc = new Scanner(*System.in*)	public void main(… sc = new Scanner();	Reserved word *Static* is absent Parameter is absent
IV	TestBikes.java MountainBike.java Bicycle.java	11 20 28	1 0 0	public *static* void main(… - -	public void main(… - -	Reserved word *Static* is absent - -
V	AnonymousDemo.java Age.java	13 6	2 0	public *static* void main(… } -	public void main(… }; -	Reserved word *Static* is absent Unnecessary parentheses -
VI	ArrayListTOArray.java	18	2	public *static* void main(… String array[] = new …	public void main(… String array = new …	Reserved word *Static* is absent Missing brackets in array criation.

VII	Test.java	19	4	//Java program to …
				public *static* void main(…
				b = (byte)(b * 2);
				println("Prefix = " + ++i);
					-
					public void main(…
					b = (byte)(2 * b);
					println("Prefix = " +++i);
						Missing comment
						Reserved word *Static* is absent
						Wrong order of the elements
						Missing space

Total		217	16			

Pérez-Castillo et. al [17] proposed a modernization process called *Data Contextualization* technique that takes as an input a legacy system with embedded SQL queries and generates as an output a model with the database schema. In the first step several SQL queries embedded in the source code written in Java programming language are represented in a extended KDM instance for supporting SQL artifacts, such as database model and SQL schemes. A static analysis is performed in the KDM instance in order to recognize SQL queries and generate a SQL statement model which is an instance of SQL-92 metamodel. Finally, several QVT rules generate the output of the process because they transform the SQL statement model into a Database schema model.

As the authors propose an entire modernization cycle, possibly they have developed a TE that takes as input KDM and generates a SQL Statement Model, which is a PSM. However, they do not provide details about the effort of this phase.

Rodríguez-Echeverría et al. [20] proposed an outline framework for the systematic process for Web Applications (WA) to Rich Internet Application (RIA) modernization. The modernization process follows the ADM approach and it is composed of 5 phases: (*i*) information extraction; (*ii*) the information extracted is stored in a KDM instance and refined with dynamic information; (*iii*) model refinement to RIA patterns, the KDM instance is improvement by finding expressions of RIA characteristics; (*iv*) model transformation, the KDM instance is now refined in to a RIA-extended Model-Driven Web Engineering (MDWE); and finally in (*v*) Converting the model instance in an executable web application.

As this approach is a proposal neither the transformations is implemented nor the strategy is deeply discussed. The authors argue that the possible solutions would be reuse some existing techniques and tools, so the whole approach does not propose a methodology to the M2M transformation.

Feliu Trias Nicolau [24] has proposed *ADMigraCMS* that defines guidelines to migrate CMS-based (CSM - Content Management Systems) Web applications to other CMS platforms supported by a toolkit. It is composed of three reengineering stages defined in the ADM *horseshoe* process and structured in four different modelling levels. The *ADMigraCMS* tool transforms automatically the transition between the levels, i.e., from PHP code-to-PHP_Model(L0), from PHP_model(L0)-to-ASTM_Model (L1), from ASTM_Model (L1)-to-KDM (L2) and from KDM (L2)-To-CMS(L3) and the inverse transformations in the forward engineering stage.

The *ADMigraCMS* tool has a complete level of automation and completes the entire Modernization cycle for PHP-implemented CMSs. Although the author has shown the mapping between the elements of different abstraction levels, it is not clear how the development of the transformation rules were performed because the implementation is not presented.

Pérez-Castillo et. al [19] proposed a declarative model transformation in order to transform KDM instances into BPMN models. The first step was to identify specific structures of meta-classes in the KDM instances and establishes other specific structures of business meta-classes in output models. The patterns are built by taking into account business patterns that are usually used by business experts for modeling business processes. The patterns also add those structures of source code elements (defined through KDM elements) that originate the specific business structures in BPMN models. The second step was to implement the model transformations by means of QVT-relation declarative language.

The authors do not detail how was the process to create the patterns and if their approach can be reused for other types of transformation. They paid more attention in the QVT-r code that shows the implementation of the patterns.

7 CONCLUSION

In this paper, we presented the approach to create KDM2PSMs transformation engine that helps modernization engineers to complete the forward engineering stage of ADM horseshoe model.

Our approach is characterized by three main features: *(i)* it uses an iterative and incremental process to develop the forward transformation rules KDM2PSM; *(ii)* it relies on analyses and comparisons of PSM instances as the main source of knowledge to develop the rules; *(iii)* it uses the *KDM-Java model mapping* as the main artifact, which is the step that establishes the mapping between the KDM and PSM metamodel elements.

The generic approach to create KDM2PSM transformation engines allow us the construction of the RUTE-K2J tool. The tool is

composed of 55 transformation rules, 28 helpers and 10 lazy rules developed with ATL, and provides an instrument to transform the KDM instance to Java Model. It can be access in the folowing URL: https://github.com/Advanse-Lab/RUTE-K2J

To demonstrate the correctness of our tool, we elaborated seven test cases that are result of a detailed evaluation process application, with the strategy of executing each transformation rule at least once. Our evaluation showed that the 92% of the source code was preserved and the information lost is mainly because the KDM input did not carried complete information along the reengineering process.

In the future, we plan to automate the approach by building a support tool for making the mapping between different metamodels and generating, semi-automatically, the transformation rules. Furthermore, we plan to improve the reverse PSM2KDM transformation engine offered by Modisco tool aiming to guaranteeing a complete KDM instance generation that preserves the information required for the use of our approach.

ACKNOWLEDGEMENT

We would like to thank the financial support provided by FAPESP, SP, Brazil, process number (2016/03104-0), CAPES and Becas Chile (CONICYT) Scholarship.

REFERENCES

[1] 2006. Eclipse Acceleo Project. http://www.eclipse.org/acceleo/. (2006). Accessed: 2018-04-28.
[2] 2006. Eclipse ATL Project. https://projects.eclipse.org/projects/modeling.mmt.atl. (2006). Accessed: 2017-11-01.
[3] 2013. Eclipse Epsilon Project. http://www.eclipse.org/epsilon/. (2013). Accessed: 2018-04-28.
[4] 2018. ADM Vendor Directory Listing. https://www.omg.org/adm/vendor/list.htm. (2018). Accessed: 2018-04-28.
[5] Franck Barbier, Gaëtan Deltombe, Olivier Parisy, and Kamal Youbi. 2011. Model driven reverse engineering: Increasing legacy technology independence. In *Second India Workshop on Reverse Engineering*, Vol. 125. 126–139.
[6] Keith Bennett. 1995. Legacy Systems: Coping with Success. *IEEE Softw.* 12, 1 (Jan. 1995), 19–23. https://doi.org/10.1109/52.363157
[7] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. 2010. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 173–174.
[8] Javier Canovas and Jesus Molina. 2010. An architecture-driven modernization tool for calculating metrics. *IEEE software* 27, 4 (2010), 37–43.
[9] Javier Luis Cánovas Izquierdo and Jesús García Molina. 2009. *A Domain Specific Language for Extracting Models in Software Modernization.* Springer Berlin Heidelberg, Berlin, Heidelberg, 82–97. https://doi.org/10.1007/978-3-642-02674-4_7
[10] Rafael S Durelli, Daniel SM Santibáñez, Márcio E Delamaro, and Valter Vieira de Camargo. 2014. Towards a refactoring catalogue for knowledge discovery metamodel. In *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*. IEEE, 569–576.
[11] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 37–54. https://doi.org/10.1109/FOSE.2007.14
[12] Object Management Group. 2009. Architecture-Driven Modernization Standards Roadmap. https://www.omg.org/adm/ADMTF%20Roadmap.pdf. (2009). Accessed: 2018-01-15.
[13] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositoriesa (MSR)*. ACM, 92–101.
[14] André de Souza Landi, Fernando Chagas, Bruno Marinho Santos, Renato Silva Costa, Rafael Durelli, Ricardo Terra, and Valter Viera de Camargo. 2017. Supporting the Specification and Serialization of Planned Architectures in Architecture-Driven Modernization Context. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 327–336.
[15] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* 152, Supplement C (2006), 125 – 142. https://doi.org/10.1016/j.entcs.2005.10.021 Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).
[16] Jorge Moratalla, Valeria de Castro, Marcos López Sanz, and Esperanza Marcos. 2012. A Gap-Analysis-Based Framework for Evolution and Modernization: Modernization of Domain Management at Red. es. In *SRII Global Conference (SRII), 2012 Annual*. IEEE, 343–352.
[17] Ricardo Perez-Castillo, Ignacio Garcia-Rodriguez de Guzman, Orlando Avila-Garcia, and Mario Piattini. 2009. On the Use of ADM to Contextualize Data on Legacy Source Code for Software Modernization. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE '09)*. IEEE Computer Society, Washington, DC, USA, 128–132. https://doi.org/10.1109/WCRE.2009.20
[18] Ricardo Pérez-Castillo, Ignacio García Rodríguez de Guzmán, Rafael Gómez-Cornejo, Maria Fernandez-Ropero, and Mario Piattini. 2013. ANDRIU. A Technique for Migrating Graphical User Interfaces to Android (S).. In *SEKE*. 516–519.
[19] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. 2010. *Implementing Business Process Recovery Patterns through QVT Transformations.* Springer Berlin Heidelberg, Berlin, Heidelberg, 168–183. https://doi.org/10.1007/978-3-642-13688-7_12
[20] Roberto Rodríguez-Echeverría, José María Conejero, Pedro J. Clemente, Juan C. Preciado, and Fernando Sánchez-Figueroa. 2012. *Modernization of Legacy Web Applications into Rich Internet Applications.* Springer Berlin Heidelberg, Berlin, Heidelberg, 236–250. https://doi.org/10.1007/978-3-642-27997-3_24
[21] Daniel San Martín Santibáñez, Rafael Serapilha Durelli, and Valter Vieira de Camargo. 2015. A combined approach for concern identification in KDM models. *Journal of the Brazilian Computer Society* 21, 1 (2015), 10.
[22] Harry M Sneed. 2005. Estimating the costs of a reengineering project. In *Reverse Engineering, 12th Working Conference on*. IEEE.
[23] Feliu Trias, Valeria de Castro, Marcos López-Sanz, and Esperanza Marcos. 2014. A Toolkit for ADM-based Migration: Moving from PHP Code to KDM Model in the Context of CMS-based Web Applications. (2014).
[24] Feliu Trias, Valeria de Castro, Marcos Lopez-Sanz, and Esperanza Marcos. 2015. Migrating Traditional Web Applications to CMS-based Web Applications. *Electron. Notes Theor. Comput. Sci.* 314, C (June 2015), 23–44. https://doi.org/10.1016/j.entcs.2015.05.003
[25] Olegas Vasilecas and Kestutis Normantas. 2011. Deriving business rules from the models of existing information systems. In *Proceedings of the 12th International Conference on Computer Systems and Technologies*. ACM, 95–100.
[26] Christian Wagner. 2014. *Model-Driven Software Migration: A Methodology* (1 ed.). Springer Vieweg. https://doi.org/10.1007/978-3-658-05270-6
[27] Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.
[28] Christian Wulf, Sören Frey, and Wilhelm Hasselbring. 2012. A Three-Phase Approach to Efficiently Transform C# into KDM. (2012).