International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-024-00735-4

FOUNDATION FOR MASTERING CHANGE

Regular

A process for creating KDM2PSM transformation engines

Guisella Angulo[1] · **Daniel San Martín**[2] · **Fabiano Ferrari**[1] · **Ignacio García-Rodríguez de Guzmán**[3] · **Ricardo Perez-Castillo**[3] · **Valter Vieira de Camargo**[1]

Accepted: 2 January 2024

Abstract

Architecture-Driven Modernization (ADM) is a special kind of reengineering that employs models along the process. The main ADM metamodel is the Knowledge Discovery Metamodel (KDM), which is a platform-independent metamodel able to represent several views of a system. Although a lot of research is currently focused on the reverse engineering phase of ADM, little has been devoted to the forward engineering one, mainly on the generation of Platform-Specific Models (PSMs) from KDM. The forward engineering phase is essential because it belongs to the end of the horseshoe model, completing the reengineering process. Besides, the lack of research and the absence of tooling support in this phase hinder the industrial adoption of ADM. Therefore, in this paper, we present a process for creating Transformation Engines (TEs) capable of transforming KDM instances in a chosen PSM. We highlight two main contributions in this work. The first is a process that software engineers can follow for building TEs capable of generating PSM instances (e.g., Java model, Python model, etc.) from KDM instances. Having that on their hands, modernization engineers can then use generators for generating language-specific source code from the PSM. The second is delivering a specific TE called RUTE-K2J, which is able to generate Java models from KDM models. The transformation rules of RUTE-K2J have been tested considering sets of common code structures that normally appear when modernizing systems. The test cases have shown that in this version of RUTE the transformation rules are able to correctly generate 92% of the source code submitted to the transformation.

Keywords Model transformation · KDM · PSM · Java model · Software modernization · Code generation

✉ G. Angulo
guisella.angulo@estudante.ufscar.br

D. San Martín
daniel.sanmartin@ucn.cl

F. Ferrari
fcferrari@ufscar.br

I. García-Rodríguez de Guzmán
Ignacio.GRodriguez@uclm.es

R. Perez-Castillo
Ricardo.PdelCastillo@uclm.es

V. Vieira de Camargo
valtervcamargo@ufscar.br

[1] Computing Department, Federal University of São Carlos, São Carlos, SP, Brazil

[2] Engineering School, Catholic University of the North, Coquimbo, Chile

[3] Institute of Technology and Information Systems, University of Castilla – La Mancha, Ciudad Real, Spain

1 Introduction

Software systems are acknowledged as *legacy* when they exhibit two main characteristics: (i) high maintenance costs (effort/time/resources) and (ii) being still essential to support current business processes. These systems cannot be discarded since they retain mission-critical business knowledge incorporated along years of maintenance [1]. For many years, traditional software reengineering has been used as a solution to this challenge, since it retains all the knowledge of these systems. However, it is well known that 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 tools [2, 3].

In 2003 the Object Management Group (OMG) started the elaboration of the Architecture-Driven Modernization (ADM). The main idea was to define standards for the reengineering process to promote industry consensus on modernization solutions and elevating the success in modernization projects.

Many companies have demonstrated interest in the ADM philosophy, as can be seen in the ADM vendor directory

Published online: 26 January 2024

listing website [4]. There are several IT companies listed as ADM partners. Some companies are specialized in modernizing systems, whereas others offer this type of service in their portfolio of solutions.

The most important ADM metamodel is the Knowledge Discovery Metamodel (KDM) [5]. Its goal is to capture and represent system architecture details in a platform and language-independent way. One of the main problems KDM intends to solve is avoiding the proliferation of different metamodels for representing legacy systems. It intends to be a standardized form of representing legacy systems, instead of letting modernization engineers free for employing different metamodels.

Figure 1 shows the horseshoe reengineering cycle and how KDM can be applied in three steps. In the reverse engineering, a legacy system is parsed into a KDM instance that represents it. In the restructuring phase the KDM instance is refactored to solve some identified problems or even to have its structure improved, resulting in a new modernized KDM instance. In the forward engineering phase the modernized KDM is used as input for generating the system, completing the modernization cycle. Within this last phase, there are still two steps, the generation of a PSM instance from the KDM and the generation of the source code from the PSM. In this paper, we focus on the former.

As previously said, most of current research on ADM has concentrated on the reverse engineering phase of ADM. Some examples are MoDisco [6], Gra2Mol [7], and Three-Phase Approach [8]. However, little research has been conducted on forward engineering from KDM. There is 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 [9, 10].

Therefore, to fill this research gap, we have developed a process for supporting modernization engineers in the creation of transformation engines (TE) that transform KDM instances to instances of some PSM, such as Java models, Python models, and others. In this work, we consider that "modernization engineers" are responsible for creating tools for supporting software modernization processes. In contrast, a software engineer is responsible for using the created tools.

The proposed process has five iterative and incremental phases, where in each cycle, one transformation rule is developed or evolved. This process emerged from the experience in creating a TE called RUTE-K2J that takes a KDM instance as input and automatically generates a Java model from it. An important point is that RUTE-K2J contributes from the second to the last step of ADM horseshoe model, opening many research possibilities for researchers and also companies in the industry to assess the potential of ADM regarding reusability, effectiveness, etc.

RUTE-K2J was evaluated with test cases in order to guarantee the 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 engine) and the Acceleo tool for generating source code from the Java model (output of the RUTE-K2J engine). So, we compare the code generated by Acceleo allowed to the original source code with aims at evaluating the correctness of the transformation. Our evaluation showed that 92% of the original source code submitted was preserved through the transformation.

In this paper, we present the following additional contributions compared to our previous conference paper [11] (Angulo et al. 2018): i) a complete rewriting of the whole paper; ii) inclusion of new activities and phases; and iii) extension of the related work and discussion sections.

The remainder of this paper is organized as follows. Section 2 presents necessary background information related to ADM and KDM and model transformations. Then Sect. 3 presents the guidelines for creating KDM2PSM transformation engines. In Sect. 4, we present the RUTE-K2J Transformation Engine. In Sect. 5, we present the validation of RUTE-K2J. Section 6 brings some discussions about our approach and our results, and Sect. 7 summarizes related work. Finally, Sect. 8 draws some conclusions and describes plans for future work.

2 Background

2.1 ADM & KDM

In 2003, OMG proposed the Architecture-Driven Modernization (ADM), an effort to standardize the process of modernization of legacy systems using models [3]. ADM promotes system modernization based on the use of models at different abstraction levels [12]: CIM (Computation Independent Model, PIM (Platform-Independent Model), and PSM (Platform-Specific Model). CIM focuses on the business environment, describing requirements at a very abstract level, without any reference to implementation aspects. In addition, PIM is a representation of a system from the platform-independent viewpoint. It focuses on the operation of a system while hiding the details necessary for a particular platform. Finally, PSM is a technological view of a system from the point of view of a platform at a low level of abstraction. In this context, ADM proposes the use of automated transformations between the models to generate new systems from legacy systems by following a horseshoe process.

The ADM modernization cycle involves three phases shown in Fig. 1: (i) reverse engineering, (ii) restructuring, and (iii) forward engineering. In the reverse reengineering phase the knowledge is extracted from source code, and a

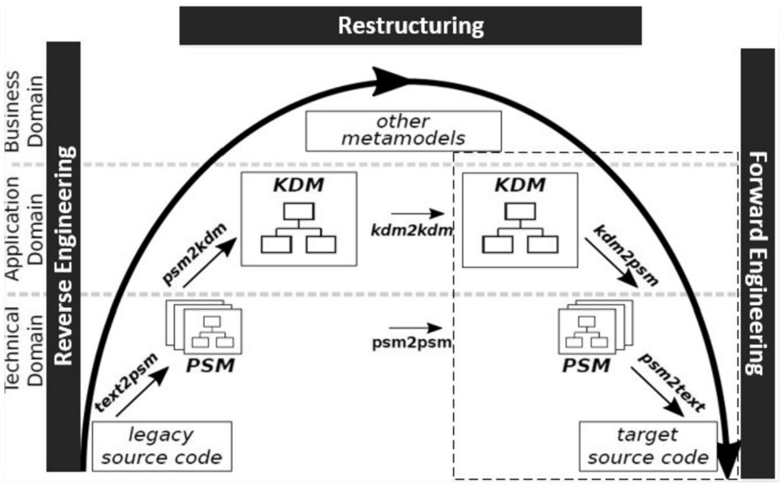

Fig. 1 Horseshoe software modernization model

PSM is generated. The PSM model serves as the basis for generating a KDM considered as PIM and CIM. Then in the restructuring phase refactorings are performed in the KDM instance that represents the legacy system to get an improved version of the system. Finally, in the forward engineering the refactored instance is transformed into source code again. The proposed process focuses on the forward engineering phase, specifically in the last two steps. Figure 1 shows the focus of this work marked with a square with dashed line border.

ADM considers seven standard metamodels, but currently only four of them have been released: Abstract Syntax Tree Metamodel (ASTM), Software Metrics Metamodel (SMM), Structured Patterns Metamodel Standard (SPMS), and KDM [3]. KDM is an OMG metamodel adopted as ISO/IEC 19506 [13, 14] 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.

Figure 2 shows the layers and packages in KDM. The Infrastructure Layer consists of the following three packages: Core, "Kdm", and Source. The Core and "Kdm" packages define together common metamodel elements that constitute the infrastructure for other packages. The Source package defines the Inventory model, which enumerates the artifacts of the existing software system and defines the mechanism of traceability links between the KDM elements and their original representation in the source code of the existing software system.

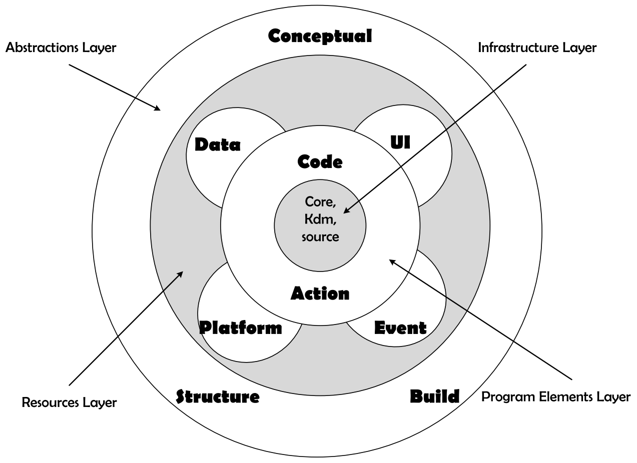

Fig. 2 Layers and packages in KDM [5]

The Program Elements Layer consists of the Code and Action packages. These packages collectively define the Code model that represents the implementation level assets of the existing software system, determined by the programming languages used in the developments of the existing software system. The Code package defines metamodel elements that represent low-level building blocks of software, such as procedures, data types, classes, and variables. Examples of these elements are ClassUnit, MethodUnit, StorableUnit, InterfaceUnit, and PrimitiveType. The Action package defines metamodel elements that represent statements as the relationship endpoints and most low-level KDM relationships, such as ActionElements, ActionFlow, and DataRelations. Examples of these elements in the package are ActionElement, Datatype, TryUnit, and CatchUnit.

The Runtime Resources Layer represents higher-level knowledge about existing software systems; this layer has the following four packages: Platform, UI, Event, and Data. Finally, the Abstractions Layer represents even higher-level

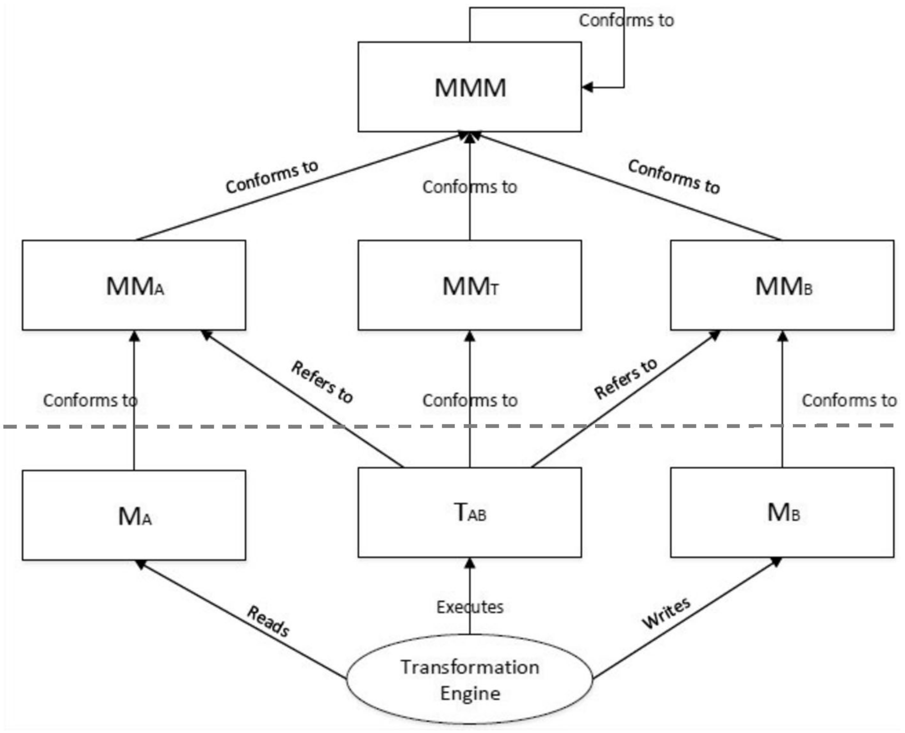

Fig. 3 Model transformation

abstractions about existing software, such as domain-specific knowledge, business rules, implemented by the existing software system, and architectural knowledge about the existing software system.

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 [15, 16].

As the work presented in this paper is totally based on model transformation, we use Fig. 3 for explaining the models and metamodels involved in this process. The figure is divided into two parts separated by a dashed line. In the lower part of the figure, it is the transformation itself where a Transformation Engine takes a model instance M_A as input, executes some transformation rules (T_{AB}) over it, and writes a new model instance M_B. That is the process that our Transformation Engine RUTE-K2J performs.

The upper part of the figure represents the metamodels, i.e., the model instance M_A conforms to a metamodel MM_A, the transformation rule conforms to a metamodel MM_T, and the model instance M_B conforms to a metamodel MM_B. It is also shown that all these metamodels (MMa, MMT, and MMb) also conforms to a metametamodel (MMM).

An important element in model transformations is the transformation rules (TRs). TRs describe how elements of a source model should be translated into elements of a target model. A transformation rule has two parts, a left-hand side (LHS) and a right-hand side (RHS). The LHS accesses the source model, whereas the RHS expands in the target model. To develop a transformation rule, it is necessary to establish a mapping between the source metamodel and the target metamodel aiming at specifying the way that target model elements must be generated from source model elements; this logic must be specified using a transformation language such as ATL and QVT (Query/View/Transformation). The process we show in this paper guides modernization engineers in creating a set of transformation rules.

Another important element is the transformation engine that executes the set of transformation rules. Each transformation rule has a purpose, i.e., transforming a specific structure. Therefore the set of transformation rules allow a complete transformation of all the structures present in the input and create an equivalent output model. At the bottom of Fig. 3, we can observe the transformation engine that reads the instance M_A, executes the set of transformation rules, and writes an instance M_B that conforms to MM_B.

Model transformation can be classified according to the metamodels that the input and output models are conformed to. Under this criterion, there are two types of transformation, *exogenous* and *endogenous*. An endogenous transformation, also called an *inplace* transformation, translates the 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* trans-

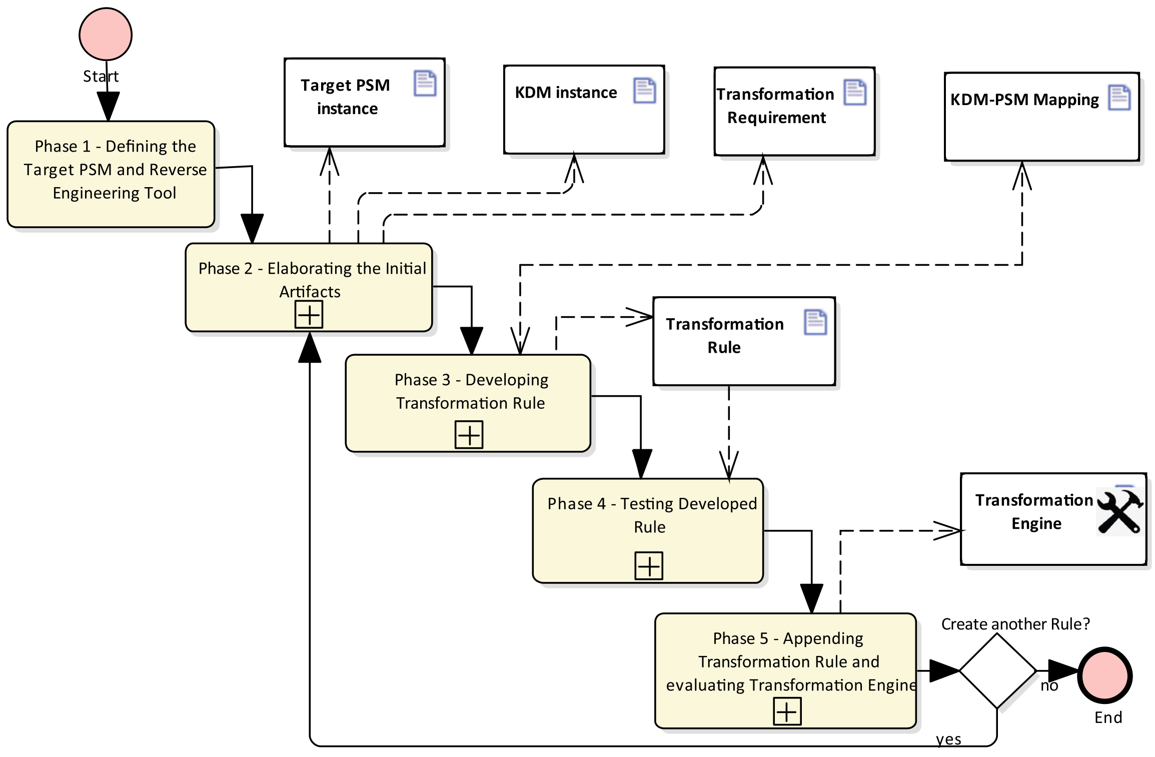
Start
Phase 1 - Defining the Target PSM and Reverse Engineering Tool
Target PSM instance
KDM instance
Transformation Requirement
KDM-PSM Mapping
Phase 2 - Elaborating the Initial Artifacts
Transformation Rule
Phase 3 - Developing Transformation Rule
Phase 4 - Testing Developed Rule
Transformation Engine
Phase 5 - Appending Transformation Rule and evaluating Transformation Engine
Create another Rule?
no
End
yes

Fig. 4 Phases of the process for creating KDM2PSM transformation engines

formation, uses different source and target metamodels, e.g., transforming a KDM instance to Java model instance [17], as we did here in this work. According to the transformation directions, a transformation can be *unidirectional* or *bidirectional*. A 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 model 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 A process for creating KDM2PSM transformation engines

This section presents our process for creating KDM2PSM transformation engines. The process is represented in Figs. 4 and 5 in different abstraction levels and in Table 1, which summarizes textually all the phases and activities. In the case of the table, it shows a more concise way to check the activities and input/output artifacts.

Figure 4 shows a more general view, focusing on the phases of the process. The return arrow (at the bottom) indicates the iterative and incremental nature of the process. It is iterative because the phases can be repeated several times until there are no more transformation requirements to be resolved. It is incremental because every time an iteration is completed, a transformation rule is created or evolved, in case it already exists. As a result, the created rule is added to the existing set of developed rules, complementing the transformation engine.

Figure 5 shows the process in a deeper way, showing the phases, activities, and the main artifacts used to complete each activity. Each following subsection details one of the phases of our process.

3.1 Phase 1 – defining the target PSM and reverse engineering tool

The first phase shown in Fig. 4 has two goals. The first one is to define which PSM (Java Model, C# Model, etc.) will be generated. It is advisable that the chosen PSM has complete and available documentation that explains its metaclasses, relationships, and all other structural details of the metamodel. This is important because these details will be needed along the process. In fact, the choice of the PSM depends on the target programming language. However, most of the time, modernization engineers know in advance in which language the source code must be generated.

The second goal is to elect a reverse engineering tool capable of generating instances of the chosen PSM from the source code. An important point here is regarding the quality of the PSM and the reverse engineering tool chosen. Regarding the PSM, there may exist different versions with

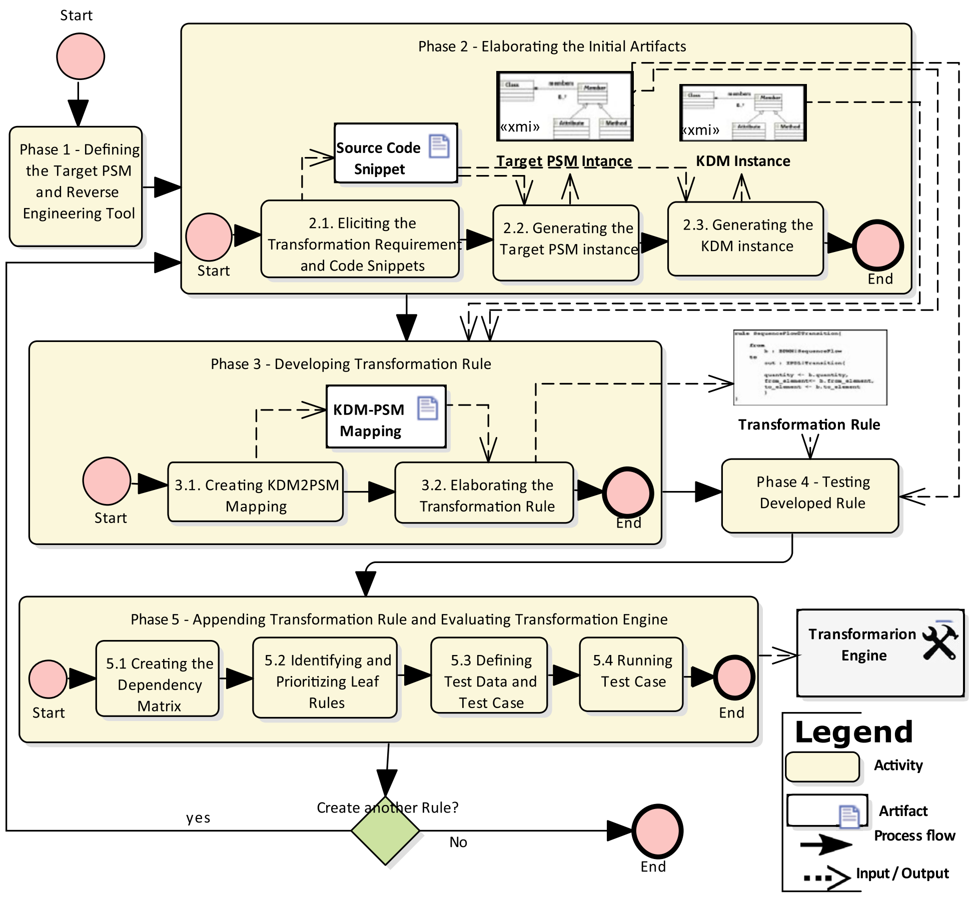
Start
Phase 1 - Defining the Target PSM and Reverse Engineering Tool
Phase 2 - Elaborating the Initial Artifacts
Source Code Snippet
«xmi»
Target PSM Intance
«xmi»
KDM Instance
Start
2.1. Eliciting the Transformation Requirement and Code Snippets
2.2. Generating the Target PSM instance
2.3. Generating the KDM instance
End
Phase 3 - Developing Transformation Rule
KDM-PSM Mapping
Transformation Rule
Start
3.1. Creating KDM2PSM Mapping
3.2. Elaborating the Transformation Rule
End
Phase 4 - Testing Developed Rule
Phase 5 - Appending Transformation Rule and Evaluating Transformation Engine
Start
5.1 Creating the Dependency Matrix
5.2 Identifying and Prioritizing Leaf Rules
5.3 Defining Test Data and Test Case
5.4 Running Test Case
End
Transformarion Engine
Legend
Activity
Artifact
Process flow
Input / Output
Create another Rule?
yes
No
End

Fig. 5 Phases and activities of the process for creating KDM2PSM transformation engine

Table 1 Summary of the phases and activities of the process

Phase name	Activities	Input	Output
Phase 1	–	–	Target PSM and Reverse Engineering tool chosen
Phase 2	**Activity 2.1** Eliciting the Transformation Requirement and Code Snippets	–	Transformation Requirement and Source Code Snippet
	Activity 2.2 Generating the Target PSM Instance	Source Code Snippet	Target PSM instance
	Activity 2.3 Generating the KDM instance	Source Code Snippet	KDM instance
Phase 3	**Activity 3.1** Creating KDM2PSM Mapping	KDM instance Target PSM Instance	KDM-PSM mapping
	Activity 3.2 Elaborating the Transformation Rule	KDM-PSM mapping	Transformation Rule
Phase 4	–	Transformation Rule	Tested transformation Rule
Phase 5	**Activity 5.1** Creating the Dependency Matrix	Transformation Rules	Dependency Matrix
	Activity 5.2 Identifying and Prioritizing Leaf Rules	Dependency Matrix	Leaf Rule Matrix
	Activity 5.3 Defining Test Data	Leaf Rule Matrix	Test Cases
	Activity 5.4 Running Test Case	Test Cases	list of differences

different characteristics and metaclasses. In the same way, there may also exist different reverse engineering tools that are able to generate PSM instances with subtle differences. So, it is important for modernization engineers to be aware of these differences and choose the one that best fits with the requirements. The output of this phase is the PSM metamodel and the reverse engineering tool chosen.

3.2 Phase 2 – elaborating the initial artifacts

The second phase has three activities shown in Fig. 5. This phase aims at eliciting the transformation requirement and elaborating the PSM and KDM instance. The activities are described in detail in the next subsections.

Activity 2.1 – Eliciting the transformation requirement and code snippets

The main purpose of this activity is to elicit transformation requirements. In the context of this work, a transformation requirement is a textual sentence of a needed transformation, i.e., it is a textual sentence of the kind of source code element that the transformation engine must be able to generate in the target PSM. An example of a transformation requirement is the following: "The transformation engine must be able to generate *If* statements with N conditions". Another example is "The Transformation engine must be able to generate *While* statements". In addition, eliciting the transformation requirements is a way of documenting the transformation engine, facilitating future maintenance and evolution tasks.

After defining the transformation requirement, the modernization engineer must implement or obtain a code snippet that represents the requirement. Sometimes it will be necessary to elaborate a complete and detailed code snippet taking into account the body, types, and relationships of the methods. In other cases, it may be necessary only to generate signatures of methods, classes, attributes, and parameters. Therefore this activity depends on the purpose of the code generation. The output of the activity is the transformation requirement and a source code snippet that represents that requirement.

Activity 2.2 – Generating the target PSM instance

This activity aims at generating a PSM instance from a code snippet obtained or elaborated in the previous activity, which is called here "Target PSM". To do that, the modernization engineer needs to use the reverse engineering tool, chosen in Phase 1, and to provide as input the code snippet developed in the previous activity. It is important to emphasize that a metamodel involves a lot of metaclasses, and in the case of the PSM instance, only few metaclasses are instantiated since they represent only the transformation requirement.

The PSM instance is an important artifact for two reasons. First, in Activity 3.1, it helps in the creation of the mapping, i.e., in the identification of the equivalent metaclasses between the KDM and the PSM metamodel. Second, in Activity 4.2 the PSM is used as a target to verify the integrity of the output PSM. Note that this PSM instance serves as an oracle, since the ideal is that the transformation rule would be able to generate an instance equivalent to the one generated here.

To facilitate the understanding of the process, we present a concrete example that involves the use of a specific PSM model. So we define the Java model as the PSM and define the following transformation requirement: "The transformation engine must be able to generate the *If* statements with one condition plus the *Else* block".Thus the modernization engineer must generate a code snippet from this requirement, and then a Java instance (PSM) is generated from this code snippet.

Listing 1 shows part of the Java instance. In line 5, we can observe the beginning of the *If-then-else* structure represented with the *IfStatement* metaclass in the Java instance. In line 8, we can notice that the *Boolean expression* of the structure is represented with the *InfixExpression* metaclass. At last, in lines 11 and 14, we can observe that the *Then* statement and the *Else* statement are represented with the *ThenStatement* and *ElseStatement*, respectively; both elements are of *Block* metaclass type.

Activity 2.3 – Generating the KDM instance

The goal of this activity is the generation of the KDM instance from the source code snippet developed in Activity 2.1. To create the KDM instance, the modernization engineer must use existing modernization tools. In the literature, there are some tools that can automate this process such as: (i) Modisco [6], which is able of generating KDM instance from Java source code; (ii) the tool proposed by Feliu Trias et al. [18], which is able of obtaining the KDM instance from PHP source code; (iii) an approach of Christian Wulf et al. [8], who proposed to transform C# programs to KDM; and (iv) the commercial software BLUAGE [19], which can transform Cobol code to KDM.

The KDM instance is an important artifact, because in Activity 3.1, it is used (with the target PSM) in the identification of equivalent-metaclasses. In addition, the KDM instance is the input of Activity 4.1, and it is used as test data to execute the transformation rule.

At this point of the process, the modernization engineer would get the code snippet that represents the transformation requirement *"If-then-else"* structure, and he will use the Modisco tool to generate a KDM instance.

Listing 2 shows part of the KDM instance. To recognize their elements with the help of the KDM metamodel documentation, we can realize that, in line 6, the beginning of the *If-then-else* structure is represented with the *ActionElement* metaclass with *kind* equal to *If*. In line 9 the *Boolean*

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <java:Model ... name="CaseControlFlow
      StatementsIf">
3   <ownedElements name="dc">
4     <ownedElements xsi:type="
      java:ClassDeclaration" ... name="
      ifDemo">
5       <statements xsi:type="
      java:IfStatement" ...>
6  ...
7  ...
8      <expression xsi:type="
      java:InfixExpression"...operator=">="
      >
9           ..
10     </expression>
11     <thenStatement xsi:type="java:Block"
        ...>
12          ...
13     </thenStatement>
14     <elseStatement xsi:type="java:Block"
        ...>
15          ...
16       </elseStatement>
17
18    </statements>
19  ...
20 </java:Model>
```

Listing 1 Target PSM of *If-then-else* structure

```
1 <kdm:Segment xmi:version="2.0" ...>
2 <model xsi:type="code:CodeModel" name="
      Case_ControlFlow
3  Statements_If">
4   <codeElement xsi:type="code:Package"
      name="dc">
5      <codeElement xsi:type="
      action:ActionElement" name="if" kind=
      "if">
6  ...
7            <codeElement xsi:type="
      action:ActionElement" name="
      GREATER_EQUALS" kind="infix
      expression">
8        ...
9          </codeElement>
10         <codeElement xsi:type="
      action:BlockUnit">
11          ...
12         </codeElement>
13         <codeElement xsi:type="
      action:BlockUnit">
14          ...
15         </codeElement>
16   </codeElement>
17  ...
18 </kdm:Segment>
```

Listing 2 KDM instance of *If-then-else* structure

expression is represented with the *ActionElement* metaclass with *kind* equal to *Infix expression*. At last, we can recognize in lines 12 and 15 the *Then* statement and the *Else* statement of the structure that are represented with the *BlockUnit* metaclass in the KDM instance.

3.3 Phase 3 – developing transformation rule

Phase 3 has two activities shown in Fig. 5. This phase has two goals: (i) to create the mapping between the KDM and PSM metamodels and (ii) to develop one transformation rule (TR). The created TR will be responsible for transforming the KDM instance that represents the transformation requirement in a PSM instance representing the same requirement.

To achieve the goals, first, the modernization engineer needs to establish a mapping between the elements (metaclasses and attributes) of the metamodels, registering each equivalence in the KDM2PSM mapping artifact. Then the modernization engineer develops the transformation rule based on this knowledge. The activities of this phase are shown in Fig. 5 and are described in more detail below.

Activity 3.1 – Creating KDM2PSM mapping

This activity aims at establishing the mapping between the PSM and KDM metaclasses. Note that the instances used for the mapping represent the same transformation requirement. Thus the output of this activity is the artifact that represents the mapping between the two different metaclasses.

To elaborate the mapping artifact, first, the modernization engineer must look for all available information with the purpose of obtaining in-depth understanding of the metaclasses, some information sources are the metamodel documentations, rules in the stage of reverse engineering (transformation from KDM to PSM), and scientific papers. The modernization engineer must review all the information by focusing on the metaclasses to increase his knowledge in the metamodels.

Second, the engineer should perform a comparative analysis between the KDM instance, output of Activity 2.3, and the target PSM instance, output of Activity 2.2. To do this, he/she must make a list of metaclasses and attributes present in the model instances. Then, with the documentation support, the modernization engineer must determine the semantic equivalence between the metaclasses and attributes in the list, establishing the correspondence between the elements. To automatize the equivalence task, the engineer may use some available tools in the literature. For example, Pérez-Castillo et al. [20] presented an ontology-based clustering technique to determine the similarity between the essential concepts in a discourse.

Finally, the modernization engineer must identify the attributes of the PSM instance that do not have an equivalent element in the KDM instance and vice versa. Due to the different levels of abstraction, the PSM model may require more detailed information than that provided by KDM. Likewise, the KDM model may contain redundant information that is not required for the PSM model. The result of this activity is the artifact called *KDM2PSM mapping*, which registers the equivalence between the KDM and PSM elements for the

Table 2 Mapping for If-then-else structure

KDM instance		JAVA model
Metaclass	Attribute	Metaclass
BlockUnit	–	Block
ActionElement	kind = “if”	IfStatement
	kind = “infix Expression”	InfixExpression

source code structure under analysis. This artifact is actively queried and updated in each iteration.

Back to the *If-then-else* example, we compare Listing 2 that represents the KDM instance to Listing 1 that represents the PSM instance. In line 6 of the Java model, we identify the equivalence between *ActionElement* metaclass with *kind=if* of the KDM instance and the *IfStatement* metaclass present in line 5 of the PSM instance. In line 9, we can establish the equivalence between the *ActionElement* metaclasses with *kind=“Infix Expression”* of the KDM instance and the *InfixExpression* metaclass in line 8 of the PSM instance. In line 11 the *BlockUnit* metaclass of the KDM instance has its equivalent in *thenStatement* with *type=Block* in line 12 of the PSM instance. In line 14, similarly, the *BlockUnit* metaclass of the KDM instance has its equivalent in *elseStatement* metaclass with *type=Blockunit* present in line 15 of the PSM instance.

As a result, Table 2 shows the equivalence between the KDM and PSM metaclasses for the *If-then-else* transformation requirement.

We must keep in mind that the proposed process is iterative and incremental. This means that, first, we need to create basic structures and then continue with the creation of more complex ones. For example, first, we need to create the Class and then create the nested methods and attributes of the Class.

Activity 3.2 – Elaborating the transformation rule

This activity aims at developing a transformation rule responsible for transforming the structure present in KDM instance to PSM instance, preserving the relationships embodied between the elements.

To develop the rule, first, the modernization engineer needs to develop the rule header. To do that, he/she needs to use the *KDM2PSM mapping* artifact and the metamodel documentation. Then he/she has to define the source and the target of the transformation, as well as the conditions to delimit the source. It is important to note that he/she must know a transformation language syntax to perform this task.

Second, the modernization engineer has to develop the rule body. To do that, he/she must place on the left side each attribute of the identified PSM metaclass assigning the equivalent KDM metaclass (on the right side), according to

```
1  rule TransformActionElementToIfStatement
     {
2   from source: KDM!ActionElement (source.
     kind='if')
3   to   target: JAVA!IfStatement (
4     originalCompilationUnit<-thisModule.
       getOriginalCompilatinUnit(source),
5     expression   <-source.codeElement,
6     thenStatement<-source.codeElement
7       -> select(e | e.oclIsTypeOf(KDM!
       BlockUnit))->first(),
8     elseStatement<-if ( (source.
       codeElement
9       -> select(e | e.oclIsTypeOf(KDM!
       BlockUnit))).size()>1)then
10          source.codeElement
11      -> select(e | e.oclIsTypeOf(KDM!
       BlockUnit))->last()
12        else Sequence{} endif
13 )}
```

Listing 3 KDM2Java Transformation rule

the *KDM2PSM mapping* artifact. Finally, one or many functions must be implemented to complete the information that cannot be obtained directly from the source KDM instance. Note that because of the low-level abstraction of the PSM model, this model needs more specific information than is provided by the KDM instance.

Continuing with the *if-then-else* example, Listing 3 shows the created KDM2Java transformation rule. In line 2, we establish as source the *ActionElement* metaclasse with the attribute *kind=if*. In line 3, we establish as target the *IfStatement* metaclass of the Java model. With the two first lines, we guarantee that each instance of the *ActionElement* with *kind=if* will have an instance of the *IfStatement* metaclass in the output Java model. In line 4 the attribute *OriginalCompilationUnit* is filled with a function that extracts the logical location of the Java class that contains the analyzed *IfStatement* structure.

In line 5, in the Java model the *Expression* attribute is assigned with the *CodeElement* element. According to the Java metamodel documentation, the *Expression* attribute can receive a set of element types including the *Infix Expression*, and for this reason, no filter is placed in the assignment. In line 6 the *ThenStatement* with *Block* metaclass type is assigned with the first *CodeElement* with *BlockUnit* metaclass type. At last, in line 8 the *elseStatement* with the *Block* metaclass type is assigned with the second *CodeElement* with the *BlockUnit* metaclass type. It is important to note that in the last two assignments, we need to put the *CodeElement* elements respecting the order of appearance given that the KDM model does not present a differentiation between the two elements, i.e., which element belongs to *then* part and which element belongs to the else part in the structure.

3.4 Phase 4 – testing the developed rule

This phase aims at testing the developed transformation rule and verifying the output model completeness. To achieve this goal, the modernization engineer has to execute the created transformation rule using some transformation tool available in literature, such as the ATL toolkit [21]. The tool must be configured to receive the KDM instance as input. Then the rule is executed obtaining as output the output PSM instance.

To verify the model completeness, the output PSM is compared with the PSM instance generated in Activity 2.2. So modernization engineers must compare both instances line by line looking for differences between the files. If any difference is found, then a structure is absent, and this will be an indicator that the transformation rule is not correctly developed, and it will be necessary to refine it by returning to Activity 3.2. The output of this activity is a list of differences between the models.

3.5 Phase 5 – appending the transformation rule and evaluating the transformation ngine

This phase aims at appending the just created transformation rule to the set of transformation rules created in previous iterations. Furthermore, this phase also aims at evaluating the transformation engine to verify if all the created TRs meet their purpose and to evaluate whether the combination of rules results in correct transformations.

Figure 5 shows Phase 5 and its four activities. These activities help to build the test cases for evaluating the transformation engine as a whole. Activity 5.1 aims at creating a matrix with all transformation rules to identify the dependencies between them. In Activity 5.2 the rules called *leaf* are identified and prioritized. In this work, a *leaf* rule is a rule that does not depend on other rules, but other rules depend on it. In Activity 5.3 the *leaf* rules are combined to select the test cases. Finally, in Activity 5.4 the test cases are executed. The activities are described with more detail below.

Activity 5.1 – Creating the dependency matrix

This activity aims at creating a dependency matrix for identifying the dependencies between transformation rules. To do that, it is necessary to create a matrix putting the rules in the rows and columns. Then in the matrix the engineer must identify the types of dependencies between the rules. The types of dependencies in this work are *mandatory*, *optional*, and *indirect*, as explained further.

- *Mandatory dependency*. It is a relationship between one rule with other rules responsible for generating the basic skeleton of the model where the new structure will be placed. Therefore the modernization engineer selects each rule in the row of the matrix and mark with the symbol "*" the mandatory dependency with the rules in the columns. Table 7 shows part of the dependency matrix; all the rules have a mandatory dependency with the rule *R01*, because this rule is responsible for generating the skeleton of the Java model. Another example is the rule *R13*, in charge of creating Fields, that has a mandatory dependency with the rules *R01*, *R05*, and *R11*. These rules are responsible for creating the PSM instance, the package, and the Class where the Fields will be placed.
- *Optional dependency*. It is a relationship between one rule with other rules responsible for creating nonbasic structures. By the term nonbasic structure we mean structures that are created nested within the main structures of the model. To recognize this kind of relationship, the engineer needs to use all the code snippets developed in Activity 2.1 (Phase 2). Then, focusing on the generated structure by the rule under analysis, he/she must identify in the code snippets all the structures that nest the structure under analysis. This kind of relationship is marked with "**".

  For example, if the rule under analysis generates the *if* structure and in the codes snippets the *if* structure is nested within the *For*, *Switch*, and *While* structures, then he/she can mark with "**" the optional dependency between the rule that generates the *if* structure and the rules that generate the *For*, *Switch*, and *While* structures. We can notice that there are many combinations of transformation rules as source code structures to represent them. So the more examples we use, the more relationships between the rules will be identified.
- *Indirect dependency*. The indirect dependency is a relationship between one rule with other rules related to the rule under analysis. This scenario is caused because there are rules that need to be executed with others rules to create a single structure. For example, the *SwitchStatement* Structure needs the execution of the "Switch-Case" rule and the "Break" rule to complete it. Therefore this type of dependency is marked with *ID-Rule*, where *Rule* indicates the rule under analysis.

The result of this first activity is the dependency matrix with all the dependencies between the rules identified.

Activity 5.2 – Identifying and prioritizing *leaf* rules

This activity aims at identifying and prioritizing *leaf* rules of the matrix elaborated in the previous activity. A *leaf* rule is a rule that does not depend on no one else. Making an analogy with a graph structure, it is a node that has no children.

Leaf rules are important to be identified because they represent a path in which these rules are the end of the path. Therefore, when we are selecting a leaf rule, we are also selecting all the other rules that belong to its path. The purpose of using *Leaf* rules is ensuring that all transformation rules are executed at least once. In other words, executing a leaf rule implies the execution of a large set of other rules beforehand.

To identify the *Leaf* rules, the modernization engineer must count the number of dependencies (marks) in each column in the matrix and select the rules with result equal to zero. Then a new matrix is generated with the identified *Leaf* rules, putting them as rows and keeping all the transformation rules in the columns. Then, for the prioritization part, considering the created *Leaf* rule matrix, the modernization engineer has to count the dependencies (marks) for each row and order them by highest to lowest. The output of this activity is the prioritized *Leaf* rule matrix.

Activity 5.3 – Defining test data and test cases

The goal here is selecting data to test the transformation engine. For this work, test data are small programs containing the source code structure that is addressed by the transformation rule (or a set of them) we want to test. For example, for testing a transformation rule that deals with an *if-then-else* structure, we must choose a piece of code that involves this conditional structure.

Our testing strategy is to work with all the identified and prioritized *leaf* rules. The execution of one prioritized *leaf* rule will also result in the execution of all the rules that depend transitively on it. Thus the combination of prioritized rules ensures that all the developed transformation rules are executed using the smallest number of test cases.

Therefore, when searching for source code to serve as test cases, every program must met the following criteria:

- *Containing the target code structure.* The source code programs must contain the code structure that are addressed by a *leaf* rule;
- *Combining leaf rules* when possible, the candidate source code program must contain more than 1 code structure with aims at testing more than one *leaf* rule at the same time.
- *Being an open-source program.* Randomly selected open-source programs hosted in GitHub repositories.
- *Being simple but a representative program.* Because it is an incremental process, programs must contain only the structures for which there are transformation rules. The complexity level of the chosen program should increase as the process continues to iterate.

After having elected the small programs that compose the test data, we need to create the set of test cases. In this work, each test case involves:

(i) **The input**, composed by one Java program *Jio* and one KDM instance *KDMi* generated from this Java program. Notice that this Java program serves also as the expected result (oracle);
(ii) **One leaf rule** *TR* to be tested;
(iii) **The oracle**, the same Java program elected as input *Jio*.

Activity 5.4 – Running test cases

This activity aims at running each created test case to evaluate the transformation engine. In general, the testing cycle is the following:

1. Electing a test case *TCi* to run;
2. Running the transformation engine given as input to the KDM instance that corresponds to the chosen program (*KDMi*) and obtaining a PSM as output.
3. Generating a Java source code from the PSM by using a code generation tool to do that (e.g., Acceleo tool);
4. Comparing the original program (which is also the oracle) with the source code generated by the code generation tool. If the output is equivalent to the oracle, then the test case succeeds. Here the user can also use the tools to compare the source code to semiautomate the task.

This cycle repeats until all the prioritized *leaf* rules are covered by test cases. The execution of the test cases shows the degree of preservation of the information expressed in lines of code, raising indications of the quality of the transformation and, consequently, the quality of the transformation rules that make up the created engine.

4 RUTE-K2J: a transformation engine from KDM to Java models

In this section, we present RUTE-K2J, a transformation engine for generating Java models from KDM models. The experience of building this tool was fundamental for creating and documenting the process presented in Sect. 3.

To develop RUTE-K2J, we used the following technologies: a the ATL transformation language [21]; b) OCL (Object Restriction Language) for data types and declarative expressions; c) Eclipse Modeling Framework (EMF); d) Modisco and the discoverers that allow the generation of the Java Model and KDM from the source code; and e) Modisco-Add, which includes improvements made in the ATL rules to generate a refined KDM instance. This first version of the *RUTE-K2J* is composed for 55 transformation rules, 28 Helpers, and 10 Lazy rules. These and other artifacts that compose RUTE-K2J are described in what follows.

i) The *KDM-PSM mapping artifact*. This artifact shows the mapping between the metaclasses of KDM and Java model. This mapping guides the development of the transformation rules, because it describes the input KDM metaclass and the equivalent Java metaclass. Table 3 shows part of *KDM-PSM mapping* artifact. The first and second columns refer to the KDM metaclass and the condition used as filter, and the third column refers to the equivalent Java metaclass. In Table 3, 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 the *Kind* attribute

Table 3 Partial KDM2Java model mapping

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

of the KDM metaclass. KDM presents many elements where the only distinction between them is the value of the *kind* attribute.

ii) The *KDM2JAVA Transformation Rules Inventory*. This artifact shows the inventory of 55 ATL transformation rules. The set of rules composes the core of the RUTE-K2J tool and allows the transformation of KDM instances to Java models. Table 4 shows part of the *KDM2JAVA Transformation Rules inventory* artifact. We can observe that the rule name makes reference to the origin and destination metaclass of the transformation. For example, the rule *CodeModelToJavaModel* has a goal to transform the *CodeModel* metaclass to the *Java model* metaclass. This rule is the principal for structuring the model and to articulate the other rules.

iii) The *ATL Helper Inventory*. In the ATL context the helpers can be viewed as functions when compared with traditional programming. The helpers allow us to factorize code out and develop a clean code. They can be called by rules or by other helpers from different points of the ATL code. For this work, we developed 28 ATL helpers. Table 5 shows part of the Helper inventory. In Table 5, we can observe the Helper *getOrphanTypes*, which allows us to obtain a sequence of *datatypes* elements present in the source Model.

iv) The *Lazy Rules Inventory*. In our project the lazy rules are used in the characterization of element collections; it must be explicitly invoked by the transformation rule. For this work, we developed 10 ATL lazy rules. Table 6 shows part of the lazy rules inventory. In the table, we can see, for instance, *setOrphanTypes* lazy rule that helps to assign values for each attribute returned by the *getOrphanTypes* Helper.

The complete RUTE_K2J artifacts[1] are complete and available for review or extension.

5 RUTE-K2J evaluation

In this section, we present an evaluation of RUTE-K2J according to the Phase 5 of our approach (Sect. 3.5). The goal was to check the correctness of the set of transformation rules and whether the combination of them results in correct transformations, i.e., whether they are correctly transforming KDM model elements into Java model elements.

5.1 Scoping

We employ the GQM (Goal–Question–Metrics) [22] template to define the scope of this evaluation. Therefore the scope of our study can be summarized as follows:

Analyze all the transformation rules of RUTE-K2J
for the purpose of evaluation,
with respect of the correctness of generated model elements,
from the point of view of software engineers conducting modernization projects, and
in the academic **context.**

[1] https://github.com/Advanse-Lab/RUTE-K2J/tree/master/Artefactos%20RUTE_K2J.

Table 4 Partial transformation rules

№	Rule name and 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

Table 5 Partial helpers

№	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 6 Partial lazy rules

№	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

5.2 Conducting the evaluation

The evaluation we have conducted followed the four activities shown in Sect. 3.5:

- **1 Creating the dependency matrix**
  As previously commented, the dependency matrix aims at supporting the identification of the dependencies among the existing rules. This matrix helps in the identification of paths of transformation rules to be executed.

Part of the dependency matrix can be seen in Table 7, as well as some dependencies. For instance, Rule R30 has mandatory dependencies with R01-JavaModel, R05-JavaPackage, R11-ClassDeclaration, R15-Method Declaration, and R16-BlockUnit. These rules are responsible for the creation of the Java model structure.

Moreover, R30 also has an optional dependency with R14-Constructor, R18-ReturnStatement, or R21-SwitchStatement. Finally, R30 has indirect dependencies with R22-BreakStatement and R23-SwitchCase marked

Table 7 Dependency matrix

Rule	R01	R05	R11	R14	R15	R16	R18	R20	R21	R22	R23	. . .
R13[1]	*	*	*	–	–	–	–	–	–	–	–	. . .
R29[2]	*	*	*	**	*	*	–	–	–	–	–	. . .
R30[3]	*	*	*	**	*	*	**	–	**	ID-R21	ID-R21	. . .
. . .	. . .	. . .	. . .	. . . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
Total	51	47	42	34	38	38	5	4	8	7	7	. . .

[1]TrasformStorableUnitToFieldDeclaration

[2]TransformWritesToSingleVariableAccess

[3]TransformReadsToSingleVariableAccess

Table 8 *Leaf* rule prioritization

Rule	R01	R05	R11	R14	R15	R16	R18	R20	R21	R22	R23	. . .	O
R47[1]	*	*	*	**	*	*			ID-R23	ID-R23	**	. . .	25
R30[2]	*	*	*	**	*	*	**		**	ID-R21	ID-R21	. . .	22
R48[3]	*	*	*	**	*	*			ID-R23	ID-R23	**	. . .	22
R50[4]	*	*	*	**	*	*	**	**	ID-R23	ID-R23	**	. . .	22
R49[5]	*	*	*	**	*	*			ID-R23	ID-R23	**	. . .	20
R31[6]	*	*	*	**	*	*						. . .	15
R51[7]	*	*	*	**	*	*						. . .	11
R52[8]	*	*	*	**	*	*						. . .	10
R46[9]	*	*	*		*	*						. . .	7
R10[10]	*	*	*		*							. . .	6
R13[11]	*	*	**									. . .	6

[1]TransformValueToNumberLiteral

[2]TransformReadsToSingleVariableAccess

[3]TransformValueToStringLiteral

[4]TransformValueToBooleanLiteral

[5]TransformValueToCharacterLiteral

[6]TransformateAddressesSingleVariableAccess

[7]TransformActionElementToNullLiteral

[8]TransformActionElementToThisExpression

[9]TransformActionElementToSuperMethodInvocation

[10]TransformInterfaceUnitToAnnotationTypeDeclaration

[11]TrasformStorableUnitToFieldDeclaration

with *ID-R21*. This means that if R30 depends on R21, then it would indirectly depend on the execution of R22 and R23. R21, R22, and R23 together compose the *Swith-Statement* structure.

The complete dependency matrix[2] is complete and available for review or extension.

- **2 Identifying and prioritizing *leaf* rules**

[2] https://github.com/Advanse-Lab/RUTE-K2J/tree/master/Test%20Data.

This activity focuses on the identification of the leaf rules. As previously commented, leaf rules help in the execution of many other rules that belong to the path this leaf rule belongs to. For our evaluation, we identified 11 leaf rules. Table 8 shows the prioritized leaf rules, which are listed in the first column of the table. It is possible to see in the last column that the execution of the rule R47 leads to the execution of 25 transformation rules (in the best case) including R01, R05, and R11, among others.

Table 9 Test cases X leaf rules

Test Cases	R47	R30	R48	R50	R49	R31	R51	R52	R46	R10	R13	C	P%
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

- **3 Defining test data and test cases**

  In this activity, we need to select a set of small programs to serve as our test data. This selection was guided by the eleven leaf rules that prioritized in the previous activity (Table 8). Therefore we selected, in most of the cases, one program per a combination of leaf rules. As a result, we ended up with seven test cases following the definition we have presented in Activity 5.3 of Sect. 3.5.

  Table 9 shows the relationship between test cases, leaf rules, and the total number of rules executed by the test case. For example, test case I executes part of the leaf rule R47, i.e., 10 rules from the set of rules that depend on this leaf rule. Also, it executes part of R30 (7 rules) and R51 (2 rules). This means that this test case executes 19 rules (36.54%) out of the 52 available.

  Note that the test cases execute the same set of mandatory rules to conform the software program skeleton, as well as they can execute already exercised structures. In that context, we count the executed rules only once. For example, test case VII executes only four new rules.

  The complete test data[3] are available for review or extension.

- **4 Executing test cases**

  The execution of the test cases followed the testing cycle we have defined previously. Therefore we have generated the KDM instance for each program. After that, we executed the transformation engine RUTE-K2J having as input each KDM instance and obtaining as output the Java model (XMI File). Next, with the aim of verifying the completeness of the model, we used the Acceleo plug-in to generate Java source code from the Output Java model. Finally, the free tool *Pretty Diff* helped us to compare the original Java code and the Java code generated by *Acceleo*. The result of this comparison is expressed in differences in lines of code between the programs.

[3] https://github.com/Advanse-Lab/RUTE-K2J/tree/master/Test%20Data.

5.3 Evaluation results

Table 10 shows the seven test cases. The first column shows the identifier of the test cases, and the second one shows the name of the Java class. The third and fourth columns show the LOC (lines of code) of the program and the LOC-D (lines of code – differences) after the generation, respectively.

The fifth and sixth ones show the original and generated lines of the code with differences. Note that the detailed lines of code correspond to the quantity indicated in the fourth column (LOC-D). The goal of these two columns is to exhibit the errors in the generation of the source code, which allows us to correct them. The last column summarizes the kind of mistakes found.

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

- For all test cases, the reserved word *Static* is not being generated. Keeping track of the data preservation during the transformations, we found out that these data are not brought by the input KDM instance.
- Test case I. The error is related to the order of the elements in the conditional part of the *if* structure. The KDM model does not have an indication of the correct position of the elements in the structures.
- Test case III. The *System.in* parameter is absent in the creation of the *Scanner* class. This is due to the lack of information in the KDM model.
- Test case V. An unnecessary parenthesis is being generated (i.e. *"()")*. We realized that this error is displayed when an anonymous class structure is present in the source code. This is a bug in the transformation rule when working in conjunction with other rules.
- Test case VI. When creating a *Array []* structure, the "*[]*" element is absent. By tracking the data during generation we found that the KDM input model does not have this information.

Table 10 Rute-K2J evaluation

Nº	File name	LOC	LOC-D	Original Source Code	Generated Source Code	Comparison Result
I	ControlFlow-Statements	61	4	public *static* int getMonthNumber	public int getMonthNumber	Reserved word *Static* is absent
				public *static* void main(. . .	public void main(. . .	Reserved word *Static* is absent
				if (month == null)	if (null == month)	Wrong order of the elements
				if (returnedMonthNumber == 0)	if (0 == returnedMonthNumber)	Wrong order of the elements
II	TestArray	20	1	public *static* void main(. . .	public void main(. . .	Reserved word *Static* is absent
III	GetAge	21	2	public *static* void main(. . .	public void main(. . .	Reserved word *Static* is absent
				sc = new Scanner(*System.in*)	sc = new Scanner();	Parameter is absent
IV	TestBikes	11	1	public *static* void main(. . .	public void main(. . .	Reserved word *Static* is absent
	MountainBike	20	0	–	–	–
	Bicycle	28	0	–	–	–
V	AnonymousDemo	13	2	public *static* void main(. . .	public void main(. . .	Reserved word *Static* is absent
				}	}();	Unnecessary parentheses
	Age	6	0	–	–	–
VI	ArrayListToArray	18	2	public *static* void main(. . .	public void main(. . .	Reserved word *Static* is absent
				String array[] = new . . .	String array = new . . .	Missing brackets in array criation.
VII	Test	19	4	//Java program to . . .	–	Missing comment
				public *static* void main(. . .	public void main(. . .	Reserved word *Static* is absent
				b = (byte)(b * 2);	b = (byte)(2 * b);	Wrong order of the elements
				println(“Prefix = ” + ++i);	println(“Prefix = ” +++i);	Missing space
Total		217	16			

- Test case VII. There is a missing comment in the transformation output. Currently, there is no transformation rule that generates comments. Then in the multiplication statement *(2 * b)* the order of the elements is wrong; it is the same problem already reported in the *Test case I*. Finally, in the expression *"+++ i"*, it can observe a lack of separation between the "+" with "++i". We identified that the problem comes in the code generation and the use of the Acceleo Tool.

As result of the evaluation, 92% of the code was successfully generated. This shows that RUTE-K2J generates a Java model that preserves the information embodied in the source code. We recognize that the current results can lead to semantic and syntactic problems if not corrected, so this evaluation will help us to carry out two types of important corrections: (i) internal, which means the correction of the transformation rules that make up the transformation engine, and (ii) external, in relation to the tools we use to generate the KDM model from the source code and the tool to generate the source from the model; these are the Modisco and Aceleo tools, respectively. The Modisco project makes publicly available the transformation rules used to generate the KDM model. So to refine the model for our purposes, we can correct the rules that generate the structures with errors identified in the evaluation. In that regard, we have already identified and corrected errors in their transformation rules during the development of this project, which were accepted by the Eclipse team.

5.4 Threats to validity

In this section, we describe some internal and external threats that can impact on the results of our evaluation.

Internal validity. *All the combinations among rules were not exercised.* All the combinations we have created were based on the dependency matrix. Therefore we created combinations that we consider the most used by developers and those that involve basic structures. We believe that having guaranteed the correctness for basic structures, more advanced ones can be conducted later.

The manual mapping among models. One of the main steps of our approach is mapping the KDM against a PSM. Currently, this is done manually, letting this process too much depend on the skills and experience of the engineer. So this also impacts on the quality of the generated rules and clearly on the final results.

External validity. *The small number of source code examples we employed.* Although we have used 10 examples of source code, which can be considered a small number, we concentrate on getting the most used source code structures, which are largely employed by developers. More advanced structures need to be investigated in future work.

The employment of Modisco-Add and Acceleo tools. Modisco-Add aims at generating a complete KDM instance. As we said before, this tool includes corrections of some bugs found in the ATL rules of Modisco. Although the tool includes the improvements developed, it still does not generate a complete KDM instance. Regarding Acceleo, we employed it for the phase of source code generation from PSM Model (Java model); also, it was employed in the evaluation section of the RUTE-K2J tool. By using Acceleo we were able to notice that some elements in the model were not transformed. So, in future works, we must consider other alternatives of source generation tools.

6 Discussion

During the process of construction of *RUTE-K2J*, we faced some difficulties:

Abstraction level. Due to the lower level of abstraction of the target Java model, there are metaclasses that need more information than the one provided by KDM instances. For example, the *CompilationUnit* (Java metaclass) has the attributes *name*, *originalFilePath*, *types*, and *imports*. On the other hand, the metaclass KDM equivalent, the *InventoryItem* metaclass, only has the attribute *name* and *originalFilePath*. The generation of a complete model is important to avoid the loss of information along the process. Thus we had to deal with the abstraction level problems.

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. On the other hand, in the Java model the metaclasses present attributes with explicit names that indicate the right assignment order. For example, the *RightOperand* and *leftOperand* attributes of the *InfixExpression* metaclass. Therefore, for the KDM model, we deduce that the order of the elements in the model agrees with the order of appearance.

External model. The KDM has a model called *External* that stores the references of the external classes (imported classes). In addition, this model stores the initial values of the local variables declared in the method body. The combination of these different elements in the external model makes it difficult to (i) retrieve the assigned value to the variable and (ii) recognize and transform each different element in the *External* model.

Enumeration data type. One of the main problems faced with KDM is that the metamodel stores constant values in string attributes, which must be stored in data type enumeration. This characteristic makes it difficult to identify all values in advance. For example, the *ActionElement* metaclasses has a string attribute called *Kind* that stores the following values: *For*, *If*, *Switch*, etc. We were not able to identify those values beforehand; only they were identified during the mapping performed.

Control flow structure *For*. The KDM model uses the *ActionElement* metaclass with type equal *Variable Declaration* for the declaration of local variables and for the variable initialization of the *For* control structure. On the other hand, the Java model uses the "VariableDeclarationStatement" for local variables and "VariableDeclarationExpression" for variable initialization in the *For* structure. That is, a KDM metaclass (*Variable Declaration*) can turn in two different JAVA metaclasses (*VariableDeclarationStatement* and *VariableDeclarationExpression*). The difficulty here is that KDM metaclass (*Variable Declaration*) does not have any other characteristic that helps in the differentiation for the election of one metaclass or another. For this case, the implementation of a *helper ATL* that returns the container type of the variable helped us to choose the correct metaclass during the development of the transformation rules.

***Switch* control flow structure**. In the KDM model the *Switch* structure does not present differentiation between the *Case* and *Default Case* element, i.e., the *Default Case* does not present any reserved word or some type or kind that identifies it as the default value of the structure. In this case, we have to deduce that the last *case* element without internal instructions is the *Default Case* element.

7 Related work

Most related works shown here focus on highlighting some parts of the entire modernization process. Thus we present some works that present transformations from the KDM model to other metamodels. However, several details of the transformations or the process for elaborating the rules are not clear.

Pérez-Castillo et al. [23] proposed a modernization process called the *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 an extended KDM instance for supporting SQL artifacts, such as database model and SQL schemes. A static analysis is performed in the KDM instance to recognize SQL queries and generate an 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.

The authors propose an entire modernization cycle using for the process an extension of the KDM model. In this work, it is not clear how the transformations between the KDM model and the other metamodels are done.

Rodríguez-Echeverría et al. [9] 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 five 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 into an RIA-extended Model-Driven Web Engineering (MDWE); and, finally, (v) converting the model instance in an executable web application.

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

Trias et al. [10] 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* cycle and structured in four different modeling 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 present.

Pérez-Castillo et al. [24] proposed a declarative model transformation to transform KDM instances into BPMN models. The first step was to identify specific structures of metaclasses in the KDM instances and establish other specific structures of business metaclasses 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 the process to create the patterns was and if their approach can be reused for other types of transformation. They paid more attention to the QVT-r code that shows the implementation of the patterns.

Pérez-Castillo et al. [25] proposed a method for extracting Enterprise Architect (EA) models, represented using ArchiMate, from KDM models. ArchiMate metamodel allows the representation of EA from different viewpoints, considering layers and aspects as the two main dimensions.The proposed approach considers the extraction of KDM model from the source code and the transformation between the KDM model to ArchiMate Model.

The authors describe the main rules to transform the KDM model to ArchiMate Model, showing part of the ATL code. The proposal is located in the Reverse Engineering part, increasing the abstraction level in each step. On the other hand, our work focuses on Forward Engineering, reducing the abstraction level in our transformations to obtain the PSM model from the KDM model.

8 Conclusion

In this paper, we presented a process for creating transformation engines able to transform KDM into any other PSM. The process aims at assisting modernization engineers to complete the forward engineering stage of the ADM horseshoe model.

Our process is characterized by three main features: (i) it uses an iterative and incremental process to develop the forward transformation rules; (ii) it relies on analysis and comparison of PSM instances as the main source of knowledge to develop the rules; (iii) it uses the *Mapping* as the main artifact, which registers the mapping between the KDM and PSM metamodel elements.

The generic process to create KDM2PSM transformation engines guided the construction of the RUTE-K2J transformation engine. The engine 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. All the artifacts are complete and available[4] for review or extension.

To demonstrate the correctness of our tool, we elaborated seven test cases that were the result of a detailed and manual evaluation process with aims at executing each transformation rule at least once in a source code program. Our evaluation showed that 92% of the source code was preserved and the information lost is mainly because the KDM instance used as input did not conserve the complete information along the modernization process.

In the future, we plan to automate the proposed process by building a support tool to make the mappings between the different metamodels and support in the semiautomatic generation of the transformation rules. In addition, we plan to improve the reverse PSM2KDM transformation engine offered by the Modisco tool to guarantee a complete KDM instance with all the necessary information for use in our process.

[4] https://github.com/Advanse-Lab/RUTE-K2J.

Acknowledgement We would like to thank the financial support provided by FAPESP, SP, Brazil, process number (2016/03104-0) and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

Author Contribution Daniel San Martín, Fabiano Ferrari, Ignacio García-Rodríguez de Guzmán, Ricardo Perez-Castillo and Valter Vieira de Camargo contributed equally to this work.

References

1. Bennett, K.: Legacy systems: coping with success. IEEE Softw. **12**(1), 19–23 (1995)
2. Sneed, H.M.: Estimating the costs of a reengineering project. In: Reverse Engineering, 12th Working Conference on. IEEE, New York (2005)
3. Group, O.M.: Architecture-driven modernization standards roadmap (2009). https://www.omg.org/adm/ADMTF%20Roadmap.pdf. Accessed: 2022-01-15
4. ADM vendor directory listing (2018). https://www.omg.org/adm/vendor/list.htm. Accessed: 2023-01-10
5. About the knowledge discovery metamodel specification version 1.3 (2011). http://www.omg.org/spec/KDM/1.3. Accessed: 2021-08-15
6. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: Modisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 173–174. ACM, New York (2010)
7. Cánovas Izquierdo, J.L., Molina, J.G.: A Domain Specific Language for Extracting Models in Software Modernization pp. 82–97. Springer, Berlin (2009)
8. Wulf, C., Frey, S., Hasselbring, W.: A three-phase approach to efficiently transform C# into KDM (2012)
9. Rodríguez-Echeverría, R., Conejero, J.M., Clemente, P.J., Preciado, J.C., Sánchez-Figueroa, F.: Modernization of Legacy Web Applications into Rich Internet Applications, pp. 236–250. Springer, Berlin (2012)
10. Trias, F., de Castro, V., Lopez-Sanz, M., Marcos, E.: Migrating traditional web applications to CMS-based web applications. Electron. Notes Theor. Comput. Sci. **314**(C), 23–44 (2015)
11. Angulo, G., Martín, D.S., Santos, B., Ferrari, F.C., de Camargo, V.V.: An approach for creating KDM2PSM transformation engines in ADM context: the RUTE-K2J case. In: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, pp. 92–101 (2018)
12. Wagner, C.: Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems. Springer, Washington (2014)
13. ISO/IEC 19506:2012 (2017). https://www.iso.org/standard/32625.html. Accessed: 2021-08-15
14. Pérez-Castillo, R., de Guzmán, I.G.-R., Piattini, M.: Knowledge discovery metamodel-ISO/IEC 19506: a standard to modernize legacy systems. Comput. Stand. Interfaces **33**(6), 519–532 (2011)
15. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: 2007 Future of Software Engineering. FOSE ’07, pp. 37–54. IEEE Computer Society, Washington (2007)
16. Júnior, J.U., Penteado, R.D., de Camargo, V.V.: An overview and an empirical evaluation of UML-AOF: an UML profile for aspect-oriented frameworks. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2289–2296 (2010)
17. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)

18. Trias, F., de Castro, V., Lopez-Sanz, M., Marcos, E.: A toolkit for ADM-based migration: moving from PHP code to KDM model in the context of CMS-based web applications (2014)
19. Barbier, F., Deltombe, G., Parisy, O., Youbi, K.: Model driven reverse engineering: increasing legacy technology independence. In: Second India Workshop on Reverse Engineering, vol. 125, pp. 126–139 (2011)
20. Pérez-Castillo, R., Caivano, D., Piattini, M.: Ontology-based similarity applied to business process clustering. J. Softw. Evol. Process **26**(12), 1128–1149 (2014)
21. Eclipse ATL project (2006). https://projects.eclipse.org/projects/modeling.mmt.atl. Accessed: 2022-11-01
22. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin (2012)
23. Perez-Castillo, R., de Guzman, I.G.-R., Avila-Garcia, O., Piattini, M.: 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, pp. 128–132. IEEE Computer Society, Washington (2009)
24. Pérez-Castillo, R., García-Rodríguez de Guzmán, I., Piattini, M.: Implementing Business Process Recovery Patterns Through QVT Transformations pp. 168–183. Springer, Berlin (2010)
25. Pérez-Castillo, R., Delgado, A., Ruiz, F., Bacigalupe, V., Piattini, M.: A method for transforming knowledge discovery metamodel to ArchiMate models. Softw. Syst. Model. **21**(1), 311–336 (2022)