Change Propagation-Aware KDM Refactorings

Rafael Serapilha Durelli†, Fernando Chagas*, Bruno M. Santos*,
Marcio Eduardo Delamaro† and Valter Vieira de Camargo*
*Departamento de Computação, Universidade Federal de São Carlos,
Caixa Postal 676 – 13.565-905, São Carlos – SP – Brazil
Email: {fernando_chagas, bruno.santos, valter}@dc.ufscar.br
†Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: rdurelli@icmc.usp.br

***Abstract*—Architecture-Driven Modernization (ADM) is a model-driven alternative to conventional reengineering processes that relies on the Knowledge-Discovery Metamodel (KDM) as the base for the whole process. Unlike conventional metamodels, KDM is capable of putting together different system abstractions (Code, Architecture, Conceptual models) in an unique site and also retaining the dependencies among them. As it is known, central to modernization processes are the refactoring activities. However, most of existing model-based refactorings do not cope with propagation of the refactoring changes across other dependent abstraction levels, keeping all models synchronised. In this paper we present Propagation-Aware Refactorings (PARef), an approach for updating dependent models when specific elements are refactored. Our refactorings involve three main steps; the identification of all dependent elements, the refactoring of them and the propagation of changes in order to keep all the dependent models synchronised. We have conducted an evaluation that shows our refactorings reached good accuracy and completeness levels.**

I. Introduction

In 2003 the Object Management Group (OMG) created a task force called Architecture Driven Modernization Task Force (ADMTF). It aims to analyze and evolve typical reengineering processes, formalizing them and making them to be supported by models [2]. ADM advocates the conduction of reengineering processes following the principles of Model-Driven Architecture (MDA) [22][2], i.e., all software artifacts considered along with the process are models.

According to OMG the most important artifact provided by ADM is the Knowledge Discovery Metamodel (KDM). By means of it, it is possible to represent different system abstraction levels by using its models, such as source code (Source and Code models), Actions (Action model), Architecture (Structure Model) and Business Rules (Conceptual Model). The idea behind KDM is that the community starts to create parsers and tools that work exclusively over KDM instances; thus, every tool that takes KDM as input can be considered platform and language-independent, propitiating interchange among tools. For instance, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages.

Central to modernization processes are the refactorings. Refactorings are ..... However, most of existing model-based refactorings do not cope with propagation of the refactoring changes across other dependent abstraction levels, keeping all models synchronized [, , , ,]

In this paper we present Propagation-Aware Refactorings (PARef), an approach for updating dependent models when specific elements are refactored.

II. Background

In this section we provide a brief background to Architecture-Driven Modernization (ADM) and Knowledge Discovery Metamodel (KDM).

A. ADM and KDM

The growing interest in using Model-Driven Development (MDD) to manage software evolution [1]–[3] motived OMG to define the ADM initiative [4] which advocates carrying out the reengineering process considering MDD principles.

ADM is the concept of modernizing existing systems with a focus on all aspects of the current systems architecture and the ability to transform current architectures to target architectures by using all principles of MDD [5, p. 60].

Figure 1 depicts the horseshoe model (i.e., horseshoe is basically a left-hand side, a right-hand side and a bridge between the sides) which was adapted to ADM. As can be seen, this horseshoe model contains three main phases:

- **Reverse Engineering**: it takes a legacy system to be modernized as input, then the knowledge is extracted and a Platform-Specific Model (PSM) is generated. In addition the PSM serves as the basis for the generation of a Platform-Independent Language (PIM), which is called KDM;
- **Restructuring**: in this phase a set of restructuring/refactoring can be applied into a model's instance by means of model transformations;
- **Forward Engineering**: then a forward engineering is carried out and the source code of the modernized target system is generated.

In order to perform a systematic modernization as depicted in Figure 1, ADM introduces several modernization standards, among them there is the Knowledge Discovery Metamodel (KDM). KDM is an OMG specification adopted as ISO/IEC 19506 by the International Standards Organization for representing information related to existing software systems.

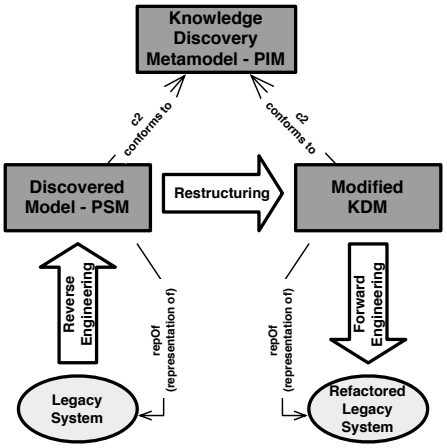

Fig. 1: Horseshoe Modernization Model. This figure is adapted from [**?**].

The goal of the KDM standard is to define a metamodel to represent all the different legacy software artifacts involved in a legacy information system (e.g. source code, user interfaces, databases, business rules, etc.).

KDM contains twelve packages and it is structured in a hierarchy of four layers: (*i*) Infrastructure Layer, (*ii*) Program Elements Layer, (*iii*) Runtime Resource Layer, and (*iv*) Abstractions Layer. These layers can be instantiated automatically, semi-automatically or manually through the application of various techniques of extraction of knowledge, analysis and transformations [4]. Figure 2 depicts the architecture of KDM and its layers. By observing this figure it is fairly evident that each layer is based on the previous layer. They are organized into packages that define a set of metamodel, whose purpose is to represent a specific and independent interest of knowledge related to legacy systems, e.g. source code, user interfaces, databases, business rules, etc.

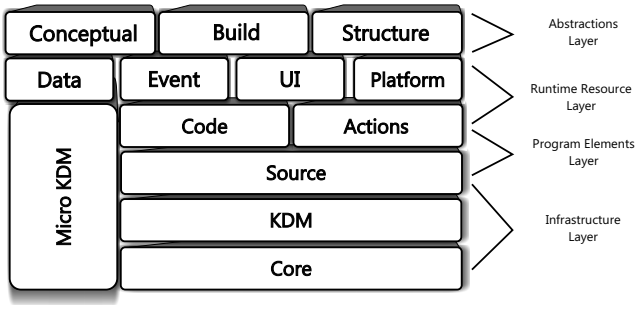

Fig. 2: KDM Architecture.

Although KDM is a metamodel to represent a whole system, its main purpose is not the representation of models related strictly to the source code nature such as Unified Modeling Language (UML). While UML can be used to generate new code in a top-down manner, an ADM-based process using KDM starts from the different legacy software artifacts and builds higher-abstraction level models in a bottom-up manner through reverse engineering techniques.

In order to show how KDM and its metaclasses can be used to represent a system, please considerer a toy system, which is depicted in Figure 3. Also, note that this system is used throughout this paper as a running example.

This toy system is based on a well know Model View Controller (MVC) design pattern. As noted in Figure 3 it is split in four KDM levels/packages, which are illustrated in the figure bounded by dashed lines shape.

The `Code Package` represents the source-code (physical artifacts). In Figure 3 is is possible to see three packages: (*i*) `GUI` ❶, (*ii*) `CTR` ❷, and (*iii*) `Model` ❸. Each of them contains a specific number of classes. Further, these classes are related to each other by means of primitive relationships, such as: `Calls`, `Creates`, `Extends`, etc - emphasized in Figure 3 by the symbol ★;

Furthermore, the `Structure Package` illustrates the system's architecture. As aforementioned the system is based on MVC - each rectangle depicts in Figure 3 represents a layer, i.e., `View` ❹, `Controller` ❺, and `Model` ❺. The View layer is realized in source-code level by the package GUI; the Model layer is realized by the package Model and the layer Controller by the CTR package. These realizations are represented in KDM by the `implementation` relationship, represented in the figure by dashed arrows. Regarding to the relationships among the layers, it is possible to visualize pipes between two layers (see Figure 3. These pipes represents the corresponding `AggregatedRelationship`[1], which represents the number summing all primitive relationships among layers. For instance, the `AggregatedRelationship` between the layer `View` and the layer `Controller` are represented by the relationships: `Calls`, `Creates`, `Extends`, and another `Calls` from the Code Package. Summing up these relationships the density value is 4. Following the same idea the relationship between the layer `Controller` and layer `Model` is 2;

Then the `Conceptual Package` illustrates the system's business rules domain. Note that this system owns three scenarios, each of them are associated with a package from Code Package by means of the association `implementation`, see the dashed arrows. Further, each scenario contains a rule except the last one. In it turn, each rule is associated with a class from Code package, again using the association `implementation`;

Finally, the `Data Package` depicts the system's database and its tables. Herein, it is possible to notice that the depicted system owns a set of Plain Old Java Objects (POJOS), they are: Student, Instructor, Secretary, and Researcher. All of these POJOS are also Object Relational Mapping (ORM), i.e., they are mapped to the Data package using the metaclasse RelationalTable.

The system described can be realized in KDM as shown in Figure 4. This figure depicts the corresponding, though simplified KDM instance of system illustrated in Figure 3. We

[1]A metaclass in KDM used to represent relationship among KDM entities.

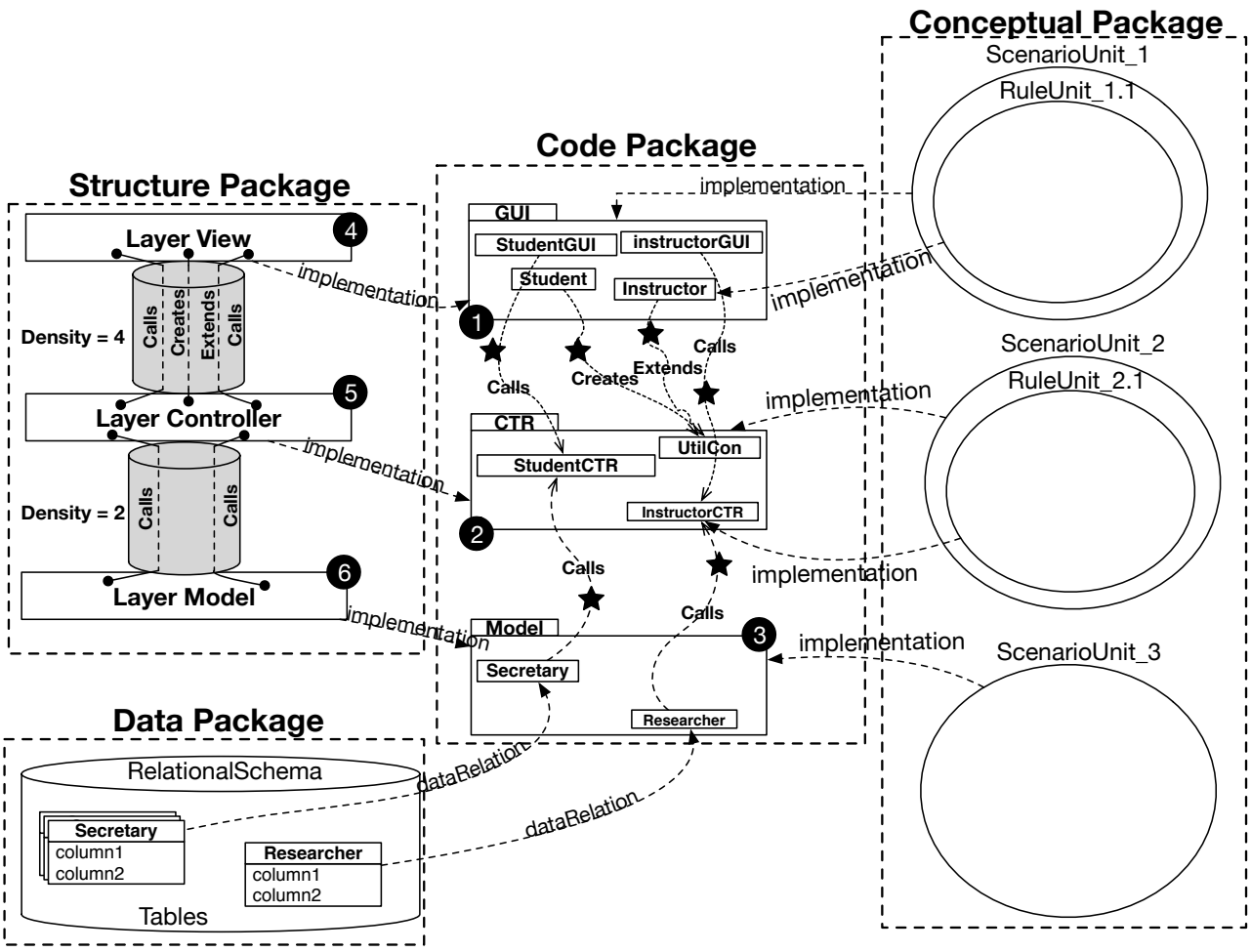

Fig. 3: Motivation and running example.

have chosen to use a KDM instance as a UML object diagram for the sake of simplicity. However, it is important to notice that due space limitations some elements are not depicted in this figure.

A KDM's instance can be understood as a tree where we have a specially node called the root of the tree. Then the remaining nodes are partitioned into $M >= 0$ joint sets $T_1, T_2, ..., T_n$, and each of these sets is a subtree. Each nodes represent a metaclass that make up the system depicted in Figure 3. The edges represent the relationship between the metaclasses. The root is the metaclass `Segment`, then, there are four subtrees rooted at `StructureModel`, `CodeModel`, `ConceptualModel`, and `DataModel`, respectively. The tree rooted at `StructureModel` has three `Layers`, `CONTROLLER`, `VIEW`, and `MODEL` - they are connected by the metaclasses `AggregatedRelationship` (see Figure 3 and Figure 4).

The tree rooted at `CodeModel` has three instance of the metaclass `Package` - `CRT`, `GUI`, and `MODEL`, respectively. Further, each package contains a set of classes, for instance, the package `MODEL` has two instance of the metaclass `ClassUnit`, `Researcher`, and `Secretary`, respectively.

The tree rooted at `ConceptualModel` also has three subtree - herein represented by the metaclass `ScenarioUnit`. Further, each node of a tree is the root of a `RuleUnit`. Finally, the `DataModel` has one subtree - `RelationalSchema`, which represent the system's data base schema. It contains four subtree - `Secretary`, `Researcher`, `Instructor`, and `Student`, where each node is an instance of the metaclass `RelationalTable`.

B. Model Transformations

Kleppe et al. [6] provide the following definition of model transformation. *A transformation is the automatic generation of a target model from a source model, according to a transformation definition*. However, according to Mens and Van [7] model transformations can be classified as either endogenous or exogenous.

Endogenous transformations are transformations between models expressed in the same language. Exogenous transformations are transformations between models expressed using different languages. Further, we can classify the transformation as: (*i*) horizontal, and (*ii*) vertical. A horizontal transformation is a transformation where the source and target models reside

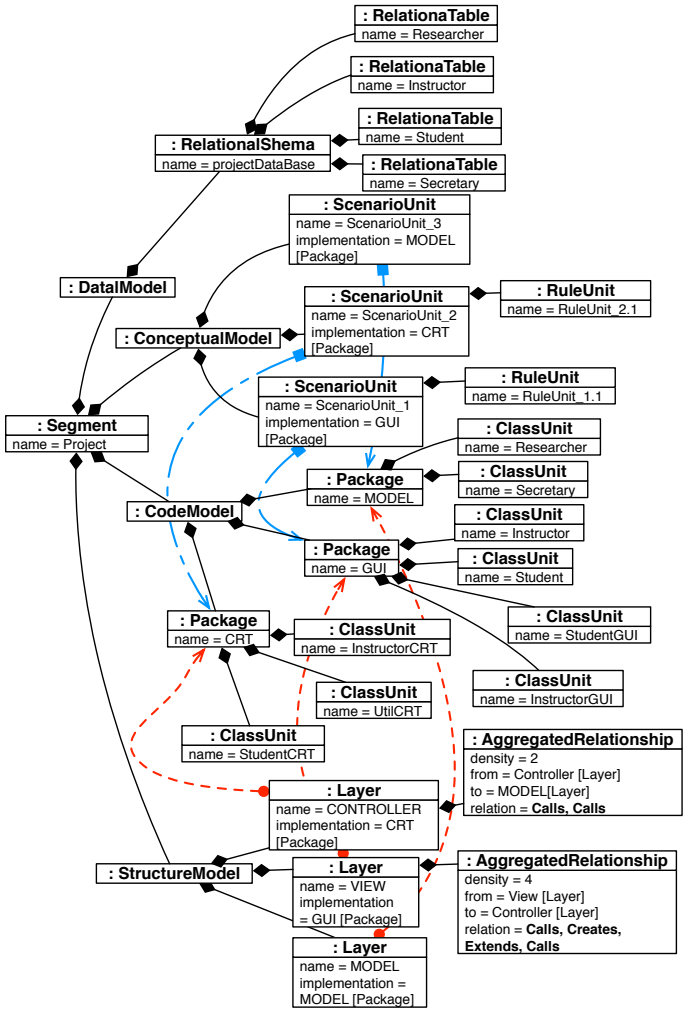

Fig. 4: A bird's eye view of a KDM's instance.

at the same abstraction level. A typical example is *refactoring*. A vertical transformation is a transformation where the source and target models reside at different abstraction levels.

TABLE I: Orthogonal dimensions of model transformations.

Classification	horizontal	vertical
endogenous	Refactoring	Formal refinement
exogenous	Language migration	Code generation

Table I illustrates that the dimensions horizontal versus vertical and endogenous versus exogenous are truly orthogonal, by giving a concrete example of all possible combinations. As can be seen in this table our approach focus in endogenous and horizontal transformations.

III. Change Propagation in KDM

In our previous work [8], we introduced a refactoring catalogue for KDM for managing evolution of a software system. This catalogue served as a starting point to investigate how the changes affect the KDM's levels.

Considering the system described earlier it is possible to remark some problem or even to add new requirements that will propagate changes at KDM's levels. For instance, a problem that can be noticed is that both classes `Student` and `Instructor` (see figure 3 ❶) should be contained in `Model` package not in `GUI` package, respectively. One way to fix this would be to apply the refactoring *Move Class*. Then both classes should be moved to the correct package. Regarding to a new requirement let's pretend someone has identified that the class `Student` is doing work that should be done by two classes, e.g., it contains attributes that hold informations upon student's addresses. In order to fulfill this new requirement one should apply the refactoring *Extract Class*. Then a new class named Address would be created (which is a POJO and also an ORM) and all student's attributes related to address would be moved to this new class.

However, these changes would rise a synchronization problem among all KDM's levels/packages. For instance, in both described refactoring it is necessary a skilled domain expert into KDM to identify all the metaclasses in the system which involve/reference the classes aforementioned and correct them respectively in all KDM packages, i.e., propagate all refactoring's impact throughout all KDM's packages/level.

In the matter of the refactoring *Move Class* (move `Student` and `Instructor` from `GUI` package to `Model` package) changes should be propagated to the `Structure Package` and to the `Conceptual Package` to maintain the model synchronized. Regarding the `Structure Package`, the `density`, i.e., aggregation relation ship between the layer `View` and the layer `Controller` would change from 4 to 2 - once the primitives relationships `Create` and `Extends` would no longer exist from the package `GUI` to the package `CTR`. On the other hand, the resulting of this refactoring would update the density between the layer `Model` and `Controller`, instead of 2 it should be 4, as `Creates` and `Extends` were also moved along with its classes, `Student` and `Instructor`. Concerning to the `Conceptual Package`, the RuleUnit_1.1 that is associated with `Instructor` should also be moved to corresponding scenario, i.e, the scenario that is associated with the package that contains now the class `Instructor` - ScenarioUnit_3.

About the refactoring *Extract Class*, the extracted class `Address` would be a POJO (it would be contained in `Model package` and it would also be an ORM - therefore, the action of this refactoring should be propagated throughout the `Data package`, i.e., the instance of `Address` should be associated with a metaclass `RelationalTable`, and its attributes should be associated with of `ColumnSet`.

These propagation seen to be easy to apply, however, in a complex system comprising all kdm's packages/levels, propagate all changes after a refactoring is a difficult and error-prone task. Even identifying the affected parts of the KDM's packages/levels is not an easy and straightforward process.

In the context of model-driven refactoring, if any change occurs at any KDM's subtree the change should be propagated to other elements. For instance, when the elements

of `CodeModel` suffer any kind of changes (e.g., are refactored), its instances, i.e., `ClassUnits`, `MethodUnits`, `StorableUnits`, etc, and related elements must be adapted accordingly so that their validity and correctness is preserved respectively. In addition, if we want to preserve others parts of KDM, like the system's structure and the business rules the `StructureModel` and `ConceptualModel` also need to adapt, respectively. In general, a change at one KDM's package/level should trigger a cascade of changes at other models. We call such sequences of adaptations change propagation.

As we can see in Figure 4, there are not only horizontally relations between the models, but the elements of the system can also be vertical related across the vertical partitions. A few examples are denoted by the red/blue dashed arrows. For instance, there is a relation between a `CodeModel` (its respective metaclasses) with the `StructureModel` - which means that a change in one of the ends of the relation can influences the other.

Considering these KDM's models as individual artifacts lead up to refactoring of each affected model separately, causing synchronization problem among them. However, this is an error-prone solution since we need to apply a refactoring at any KDM's models and then propagate all change in order to keep a KDM instance synchronized.

In order to fulfill this limitation and create an automatized process named Propagation-Aware Refactorings (PARef) that contains three main steps. The first step is the identification of all dependent elements related to a specific refactoring. In the second step the refactoring of all identified elements are performed using a model-to-model transformation language - the third step is the propagation of changes in order to keep all the dependent models synchronized. In the following sections our approach is presented.

IV. Propagation-Aware Refactorings

In order to fulfill the limitation pointed out, we introduce an approach that aims to propagate all the changes throughout the KDM's levels. This approach ensures that when a change/refactoring is performed in any KDM's level, it is correctly propagated to the affected KDM's levels and vice versa. So it make certain that the consistency between the KDM's levels when they are refactored. The described problem presented in Section III can, in our view, be split into three steps, which are depicted by its corresponding letters and tittle in Figure 5.

In the step [A], *Mine Affected Metaclasses*, we developed a mechanism which shows all metaclasses that need to be updated after applying any changes/refactoring. These metaclasses are those that have some dependence on the metaclass to be modified by the refactoring. This step is totally based on a set of queries that works on a KDM instance. In fact, this step uses depth-first search algorithm to identify all affected metaclasses along with a set of queries.

In step [B], *Apply Refactoring*, here the software modernizer has to choose an appropriate refactoring to be applied into the KDM. In this step, new metaclasses can be created, updated, and removed. Also it is necessary to gather all the needed parameters for applying the refactoring. This step uses M2M transformation language to perform the refactorings.

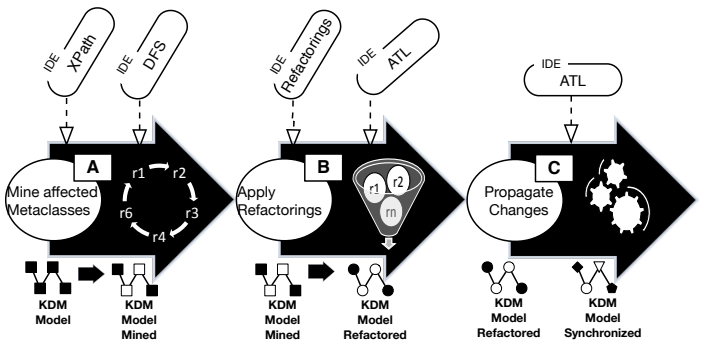

Fig. 5: Propagation-Aware Refactorings steps.

In step [C], *Propagate Changes*, involves updating the elements identified in the step [A]. As in step [B], in this step we also have used M2M to update all KDM's instances. More details on each step are provided in the next sections.

A. Mine Affected Metaclasses

The step [A] starts with a depth-first search algorithm that aims to show all metaclasses and its relationships that use somehow the metaclass(es) that will be refactored in step [B]. As input all the metaclasses that will be used to apply an specific refactoring is needed. The algorithm uses a set of queries. These queries are performed on the KDM's instance to mine all the affected/linked metaclasses. All the queries were created using XPath. We have decided to use XPath because it is a well-know and well-documented language.

Let us consider the running example depicted in Figure 3. In this example, the engineer aims to apply the refactoring *Move Class* - both classes `Student` and `Instructor` should not longer be contained in the package `View`. These classes should be allocated into the package `Model`. Considering the refactoring *Move Class*, three elements (`Student`, `Instructor`, and their package) need to be investigated throughout the KDM's instance in order to identify propagation scenarios of changes.

Therefore, firstly a query must be executed to get the root elements in KDM. This query is represented as the first statement in Figure 6, see line 1 - it is used to return an instance of the metaclass `Segment`. The returned Segment, as well as all KDM's levels are gathered by the other queries presented in Figure 6 lines 2 to 5. The returned elements of these queries are used as input in our depth-first search algorithm.

Algorithm 1 depicts the depth-first search algorithm that is used to mine all the affected metaclasses. It takes as input a KDM's instance, a `Segment`, and a set of elements that will be refactored in Step [B] (e.g., for the refactoring *Move Class* three affected elements - `Student`, `Instructor`, and their package) depicted in Figure 6. A diagram of how our depth-first search algorithm works

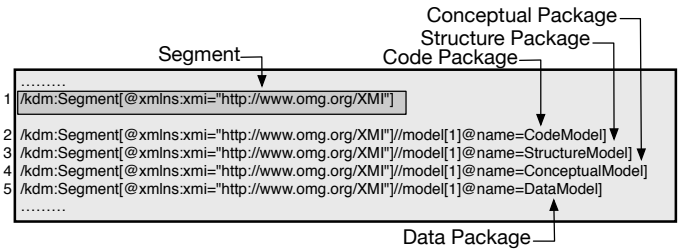

Fig. 6: Xpath used to return the KDM's root element, Segment.

Algorithm 1: DFS(G,u) - Depth-First Search Algorithm.

Input: DFS (G, u, eL) where G is a KDM's instance, u is the initial metaclass, i.e., Segment, and eL is a set of elements to verify

Output: A collection of affected metaclasses

```
1  begin
2      foreach outgoing edge e = (u, v) of u do
3          if vertex v as has not been visited then
4              if vertex v contain implementation = true
               then
5                  foreach implementations element do
6                      verify all elements in implementation
7                  end
8                  Mark vertex v as visited (via edge e).
                   Recursively call DFS (G, v).
9              end
10         end
11     end
12 end
```

is shown in Figure 7. Each node represents a metaclass and the edges represent the relationship among the meta-classes - the node A represents the `Segment` and K, H, E and B illustrate `CodeModel`, `StructureModel`, `ConceptualModel`, and `DataModel`, respectively.

More specifically, the algorithm works as follows: first it is necessary to pick a starting point, i.e., the metaclass `Segment`. Visit the `Segment`, push it onto a stack, and mark it as visited. Then it is necessary to go to the next metaclass that is unvisited, verify if it has an association named `implementation`. If yes, it verifies if this association contains references to any element's used in the refactoring, if yes - push it on the stack, and mark it. This continues until the algorithm reachs the last metaclass. Then the algorithm checks to see if the `Segment` has any unvisited adjacent metaclass. If it does not, then it is necessary to pop it off the stack and check the next metaclass. If the algorithm finds one (unvisited metaclass), it starts visiting adjacent metaclasses until there are no more, check for more unvisited adjacent metaclasses, and continue the process always verifying the association named implementation. When the algorithm finally reach the last metaclass on the stack and there are no more adjacent, unvisited metaclasses that contains the association

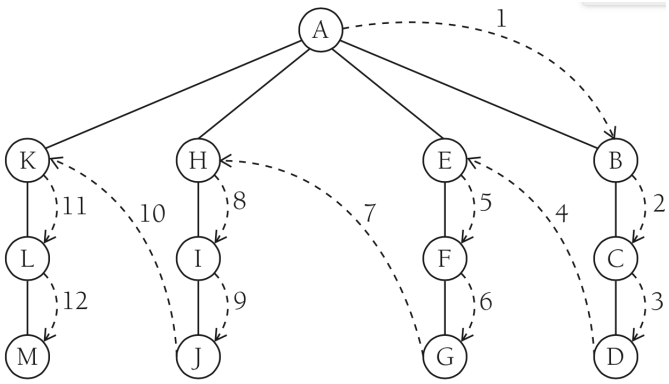

Fig. 7: Depth-First Search.

`implementation` without check , our algorithm should show a list of all affected metaclasses.

B. Apply Refactoring

In the step [B] the engineer must apply the refactoring. A natural way of implementing refactoring in models is by means of *in-place transformations*[2] as describe in Section II. Going into more details, applying these transformations/refactoring into a KDM's instance can introduce incompatibilities and inconsistencies which can not be easily resolved. In fact, we can classified these transformations by their corrupting or non-corrupting effects:

- *non-breaking changes*: changes which do not break the KDM's instance - for instance, the refactoring rename;
- *breaking and resolvable changes automatically*: changes which do break the KDM instance, but can be resolved by automatic means - for instance, apply the refactoring *move class, extract class, push meta-attributes*, etc;
- *breaking and unresolvable changes automatically*: changes which do break the KDM instance and can not be resolved automatically - for instance, when manual interventions are needed.

These transformations can be performed by means of rule-based languages. Our approach uses ATL Transformation Language (ATL). The main advantage of using one ATL is that the transformation logic can be expressed at a high level of abstraction thus enhancing maintainability and understand-ability.

To express a transformation in our approach, the user must specify mapping rules that describe how KDM's elements model can be refactored. Further, the users should inform some input parameters that should be properly instantiated. For example, considering the refactoring *Move Class*, an ATL transformation that could perform this task is depicted in Figure 8.

By inspecting this transformation/refactoring we can see important informations. For instance, lines 1 to 5 illustrate

[2]We have devised a repository where a set of *in-place transformations* (i.e., refactoring) is available. The repository can be accessed in www.site.com.br. It aims is to share refactoring to be applied into KDM's instances.

```
1  -- @atlcompiler atl2010
2  -- @nsURI MM=http://www.eclipse.org/MoDisco/kdm/code
3  -- @nsURI MM1=http://www.eclipse.org/MoDisco/kdm/structure
4  -- @nsURI MM2=http://www.eclipse.org/MoDisco/kdm/kdm
5  -- @nsURI MM3=http://www.eclipse.org/MoDisco/kdm/core
6  module moveClasses;
7  create OUT : MM, OUT1 : MM1, OUT2 : MM2, OUT3 : MM3
8  refining IN : MM, IN1 : MM1, IN2 : MM2, IN3 : MM3;
9
10
11 rule moveClass {
12  from
13       source : MM!Package (source.name = '#parameter')
14  to
15       target: MM!Package (
16            codeElement <- Sequence{thisModule.moveClassUnit('#parameter'),
17             thisModule.moveClassUnit('#parameter')... }
18       )
19  }
20
21  helper def : moveClassUnit (className : String) :
22     MM!ClassUnit = MM!ClassUnit.allInstances()->any(e | e.name = className);
```

Fig. 8: Chunk of code in ATL to perform the refactoring *Move Class*.

which KDM's level/packages are be affected by this transformation. After, in line 6 it is possible to see the name of our transformation, `moveClasses`. Lines 7 and 8 represent the output and input models that conform to the KDM, e.g., the model used during the transformation/refactoring. With the `refining` mode (see Line 8 of Figure 8), the ATL engine can focus on the ATL code dedicated to the generation of modified target elements. Other KDM elements (e.g. those that remain unchanged) are implicitly processed by the ATL engine.

Line 11 a matched rule is defined. In fact, this rule represents the refactoring *Move Class*. Occurrences of the input pattern may be filtered by introducing a *guard*, a boolean condition that KDM model must satisfy (e.g., line 13). Lines 14 though 19 the refactoring *Move Class* is actually defined. As can be seen, we are moving a set of `ClassUnit` by means of the helper (defined in lines 21 and 22). Almost all refactorings need some input parameters that should be properly instantiated by the user. For instance, consider the chuck of code written in ATL depicted in Figure 8, lines 13, 16, and 17 (`#parameter`). Therefore, before to apply the refactoring, the user should set the parameters. Considering our running example (depicted in Figure 3) the parameters would be: `Model`, `Student`, and `Instructor`, respectively. Afterward, this ATL is ready to be applied into a KDM instance.

C. Propagate Changes

In this step, all propagations regarding the refactoring *Move Class* are implemented. In fact, this step is performed along with the step [B] using a set of rules defined in ATL. In Figure 9 the ATL used to propagate the changes regarding the refactoring *Move Class* is presented. As can be seen, there are three rules - each of them is used to propagated the change in a specific KDM package, respectively. The first rule is responsible to propagate the changes throughout the `Structure Package`, see lines 24 to 32. In line 26 the source pattern of the rules is defined by using OCL [17] guard stating the layers to be matched. After, is defined a target pattern (lines 29 -31) which is used to compute the `density` of an `AggregationRelationship` after the application of the refactoring *Move Class*.

```
.....
24 rule propagationStructure {
25  from
26       source : MM1!Layer (source.allInstance
27            -> select(e | e.refImmediateComposite = '#parameter'))
28  to
29       target : MM1!Layer (
30        aggregated <- thisModule.getDensityAggregation(target.aggregated)
31       )
32 }
33 rule propagationScenario {
34
35  from
36       source : MM5!RuleUnit (source.name = '#parameter')
37  to
38       target: MM5!ScenarioUnit (
39        conceptualElement <-
40            Sequence{thisModule.getRuleUnit('#parameter')...}
41       )
42
43 }
44 rule propagationDataPackage {
45  from
46       source : MM!ClassUnit (source.allInstances()
47            -> select(e | e.refImmediateComposite = '#parameter')
48            ->collect(e | e.name = '#parameter' or e.name = '#parameter'))
49  to
50       target: MM4!RelationalTable
51       (
52            name <- source.name,
53            itemUnit <- Sequence {columns} ->
54                     union(source.codeElement->
55                          select(e | e.oclIsTypeOf(MM!StorableUnit)))
56       ),
57       columns : MM4!ColumnSet (
58            name <- '#parameter',
59            type <- if (source.type.oclIsUndefined()) then
60                          OclUndefined
61            else
62                          source.type->getType()
63            endif
64       )
65 }
....
```

Fig. 9: Chunk of code in ATL to perform the propagation after the application of refactoring *Move Class*.

The rule defined in lines 33 to 43 propagates the changes throughout the `Conceptual Package`. For instance, the `RuleUnit 1.1` that is associated with `Instructor` should also be moved to corresponding scenario, i.e, the scenario that is associated with the package that contains now the class `Instructor - ScenarioUnit 3`.

Finally, the rule defined in lines 44 - 65 aims to propagate the change to the `Data Package`. For each moved `ClassUnit`, a `RelationalTable` instance has to be created. Their names have to correspond. The `itemUnit` reference set has to contain all `ColumnSet` that have been created for each `StorableUnit` (metaclass that represent all the attributes that a class holds) as well as its types.

V. Proof-of-Concept Implementation

We devised a Eclipse plug-in named Modernization-Integrated Environment (MIE) which is split in three layers, as follows: (*i*) Core Framework, (*ii*) Tool Core, and (*iii*) Graphical User Interface (GUI). This plugin was devised on the top of the Eclipse Platform; The first layer we used both Java and Groovy as programming language. Moreover, the Core Framework layer contains a set of Eclipse plug-ins on which

our environment is based on, such as MoDisco and Eclipse Modeling Framework (EMF)[3]. We used MoDisco[4] once it is an extensible framework to develop model-driven tools to support use-cases of existing software modernization and provides an Application Programming Interface - (API) to easily access the KDM model. Also, EMF was used to load and navigate KDM models that were generated with MoDisco. The second layer, the Tool Core, is where the steps presented in Section IV were implemented. Herein, we work intensively with KDM models, which are XML files. Therefore, we use XPath to handle those types of files, to mine the affected metaclasses, ATL to perform the refactoring and to propagated them. Finally, the third layer is the Graphical User Interface (GUI) that consists of a set of SWT windows with several options to perform the refactorings based on the KDM model.

VI. Related Work

In [9], Enrico Biermann et al. propose to use the Eclipse Modeling Framework (EMF), a modeling and code generation framework for Eclipse applications based on structured data models. They introduce the EMF model refactoring by defining a transformation rules applied on EMF models. EMF transformation rules can be translated to corresponding graph transformation rules. If the resulting EMF model is consistent, the corresponding result graph is equivalent and can be used for validating EMF model refactoring. Authors offer a help for developer to decide which refactoring is most suitable for a given model and why, by analyzing the conflicts and dependencies of refactorings. This initiative is closed to the model driven architecture (MDA) paradigm [6] since it starts from the EMF metamodel applying a transformation rules.

In [10] Rui, K. and Butler, apply refactoring on use case models, they propose a generic refactoring based on use case metamodel. This metamodel allows creating several categories of use case refactorings, they extend the code refactoring to define a set of use case refactorings primitive. This refactoring is very specific since it is focused only on use case model, the issue of generic refactoring is not addressed, and these works do not follow the MDA approach.

Another work on model refactoring is proposed in [11], based on the Constraint-Specification Aspect Weaver (C-SAW), a model transformation engine which describes the binding and parameterization of strategies to specific entities in a model. Authors propose a model refactoring browser within the model transformation engine to enable the automation and customization of various refactoring methods for either generic models or domain-specific models. The transformation proposed in this work is not based on any metamodel, it is not an MDA approach.

In the line of language independent refactoring and metamodelling, Sander et al. [12], study the similarities between refactorings for Smalltalk and Java, and build the FAMIX model. It provides a language-independent representation of object- oriented source code. It is an entity-relationship model that models object-oriented source code at the program entity level, with a tool to assist refactoring named MOOSE. FAMIX does not take account neither complex features in strongly typed languages, nor aspects of advanced inheritance and genericity. This approach is not really independent from language since the refactoring transformation is achieved directly on the original code. This alternative forces to implement transformers of specific code for each language. These code transformers use an approach based on text using regular expressions.

VII. Conclusions

Acknowledgments

Rafael Serapilha Durelli would like to thank the financial support provided by FAPESP, process number 2012/05168-4.

References

[1] R. Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos, and L. Andrade, "Architectural transformations: From legacy to three-tier and services." Springer Berlin Heidelberg, 2008, pp. 139–170.

[2] L. Andrade, J. a. Gouveia, M. Antunes, M. El-Ramly, and G. Koutsoukos, "Forms2net – migrating oracle forms to microsoft .net," in *Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 261–277.

[3] T. Reus, H. Geers, and A. van Deursen, "Harvesting software systems for mda-based reengineering," in *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 213–225.

[4] R. Perez-Castillo, I. G.-R. de Guzman, O. Avila-Garcia, and M. Piattini, "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*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 128–132. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.20

[5] W. M. Ulrich and P. Newcomb, *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[6] A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[7] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, pp. 125–142, 2006.

[8] R. Durelli, D. Santibanez, M. Delamaro, and V. de Camargo, "Towards a refactoring catalogue for knowledge discovery metamodel," in *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*, 2014, pp. 569–576.

[9] E. Biermann, K. Ehrig, C. Kohler, G. Kuhns, G. Taentzer, and E. Weiss, "EMF Model Refactoring based on Graph Transformation Concepts," *Mathematics of Computation*, vol. 3, 2008.

[10] K. Rui and G. Butler, "Refactoring use case models: The metamodel," in *Proceedings of the 26th Australasian Computer Science Conference - Volume 16*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 301–308.

[11] J. Zhang, Y. Lin, and J. Gray, "Generic and domain-specific model refactoring using a model transformation engine," in *Volume II of Research and Practice in Software Engineering*. Springer, 2005, pp. 199–218.

[12] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, 2000, pp. 154–164.

[3]http://www.eclipse.org/modeling/emf/

[4]http://www.eclipse.org/MoDisco/