An Approach for Keeping Static Views Synchronized when Refactoring KDM Models

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, Business Rules, Data, Events) 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). The goal was to analyze and evolve typical reengineering processes, formalizing them and making them to be supported by models [1]. The result of this effort was the creation of Architecture-Driven Modernization (ADM), which advocates the conduction of reengineering processes following the principles of Model Driven Architecture (MDA) [2]–[4], i.e., all software artifacts considered along with the process are models. Therefore, a typical ADM-based modernization process starts with a reverse engineering phase to recuperate a model representation of the system; proceeds by applying refactorings over the recuperated model and finalize by a forward engineering phase where the modernized system is generated.

Knowledge Discovery Metamodel (KDM) is the most important metamodel provided by ADM. Its main characteristics are: i) it is an ISO-IEC standard since 2009 (ISO/IEC 19506); ii) it is platform/language independent, and ii) it is able to represent different views of the same system and retain the dependencies among them by using specific metaclasses. This third point is possible thanks to several internal KDM metamodels/packages that are focused on specific views or abstraction levels, such as source-code (Code metamodel), behaviors (Action metamodel), architecture (Structure metamodel), business rules (Conceptual metamodel), database (Data metamodel), events (Event metamodel), Graphical User Interface (GUI) (UI metamodel) and deployment (platform metamodel).

It is well known that refactoring activities are central to modernization processes. Refactorings are defined as the process of modifying the internal structure of software without changing its external observable behavior [5]. Behavior preservation in refactoring activities has received a lot of attention for years, both in source code and in models [6]–[9]. One of the known problems when refactoring models is change propagation, i.e., the modifications that need to be done in model elements that are dependent on the refactored model element. Although the behavior preservation is harder to check and characterize when dealing with models, there are works that present proposals of keeping the behavior models updated when static models are refactored [10]. Most of the works propose solutions to propagate changes across different metamodels not in the same metamodel.

However, although some research has been conducted on the theme of change propagation in models [6]–[10], none of them has devoted attention on a metamodel like KDM, which groups several metamodels under a unique place and already provide metaclasses for retaining the dependences among these models. In most cases, the related works concentrate on propagating changes in a different metamodel from where had occured the modification. Besides, the concentration of some of them are in behavior preservation, an aspect that is out of the scope of this work. Furthermore, up to this moment, few research has been done on KDM refactorings [11], limiting the dissemination and adoption of ADM. We believe that our change propagation approach will foster the creation and research on KDM refactorings.

In this paper we present an approach for propagating changes when refactoring KDM model instances. The main goal is to guarantee the global system representation keep synchronized along with the refactoring activities; which are much common during modernization processes. Our approach runs in three steps: i) a mining algorithm identifies all KDM metaclasses that need to be updated when refactoring a specific KDM metaclass, ii) an ATL Transformation Language (ATL) that performs the intended refactoring, and iii) another ATL that performs one or more model transformations that characterize change propagation. We have implemented the approach in a generic way as a decoupled module, which can be coupled to existing refactorings. In this way, existing users can write KDM refactorings in ATL without worrying about the change propagation. The only task is to provide for our component the input it needs to conduct the propagation.

The main contributions are: i) a mining algorithm to identify all KDM metaclasses that need to be updated when a specific refactoring is performed, ii) a set of refactoring devised to KDM domain, iii) a propagation technique approach, and (iv) a support and preliminary infrastructure for allowing the creation of refactorings for kdm.

This paper is structured as follows: In Section II the notion related to ADM and KDM, their details and a system's description that was instantiated in KDM are showed. In Section III a motivation is presented. Section IV shown the proposed approach. In Section V, an empirical evaluation is presented. In Section VI there are related works and in Section **COLOCAr** there are the conclusions.

II. Architecture-Driven Modernization (ADM) and Knowledge Discovery Metamodel (KDM)

According to OMG, ADM "is the process of understanding and evolving existing software assets [1]". ADM solves the problems of traditional reengineering since it carries out reengineering processes taking model-driven principles into account. However, ADM does not replace reengineering, but ADM improves it. A typical ADM process involves three main phases: Reverse Engineering, Refactorings and Optimizations and Forward Engineering. In reverse engineering phase, a legacy system is abstracted in a KDM instance. Next, some refactorings and optimizations can be applied over the KDM instance and, in the last phase, the modernized source code is generated. The focus of this paper is on the refactorings to be applied over KDM instances.

KDM is the most important metamodel of ADM (ISO/IEC 19506) and its goal is being able to represent most of the software artifacts and views into a unique place; this is very different form conventional model-driven development techniques we have found on literature [12], [13]. KDM contains twelve packages organized 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 [14]. Notice the slight difference between the terms "view" and "package" we are using in this paper. Using the KDM terminology, we can say the Code package allows the creation/existence of a view to represent the source code. At the same way, the Structure package allows the creation of an architectural view of the system.

One of the main KDM characteristics is the power to represent different abstractions of the same system and also to retain/make evident the dependencies among these abstractions. This is done thanks to its twelve internal metamodels/-packages, since each one is devoted to represent a particular view. In fact, KDM is a set of internal metamodels.

Figure 1 shows schematically the potential of KDM for representing three views of part of an Academic Management System: `Code View`, `Architecture View` and `Data View`. The entire figure represents a KDM model, i.e., an instance of KDM composed by three other internal instances - each big rectangle (packages/views) represents an instance of a internal metamodel/package. Besides, each smaller internal element (classes, packages, layers, relationships, etc) are instances of KDM metaclasses. Notice that we are using the notation "instance:Type" at every element so that the name of KDM metaclasses can be seen. As this is a MVC-based system, the `Code View` contains three instances of the `Package` metaclass: (i) `VIEW`, (ii) `CONTROLLER`, and (iii) `MODEL`. Each of them involving a specific number of `ClassUnit` instances. The classes are related to each other by means of primitive relationships, such as: `Calls`, `Creates`, `Extends`, etc.

`Architecture View` represents the system architecture. In this example each rectangle represents an instance of the `Layer` metaclass, i.e., `View`, `Controller`, and `Model`. Between Layes and Packages there is a relationship called `Implementation`, whose intention is to denote that a specific higher level abstraction is realized by one or more code elements. In this way, the `Layer View` is realized in source-code level by the package `VIEW`; the `Model` layer is realized by the package `Model` and the layer `Controller` by the `Controller` package.

KDM also possesses a relationship type called `AggregatedRelationship` whose aim is to represent dependencies between architectural elements from `Structure Package`. Using this relationship it is possible to put together several primitive relationships in a unique "channel". For example, in Figure [**?**], it is possible to see big "channel" (represented by directional arrows) between the layers schematically representing the `AggregatedRelationship`. The `AggregatedRelationship` between the layer `View` and the layer `Controller` group the following relationships: two `Calls`, one `Creates` and one `Extends`. The number aside the `AggregatedRelationship` "channel" is its "density", that represents the amount of primitive relationships the `AggregatedRelationship` involves. Summing up these relationships the "density" value is 4. Following the same idea the relationship between the layer `Controller` and layer `Model` is "density" value is 2.

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 (PO-

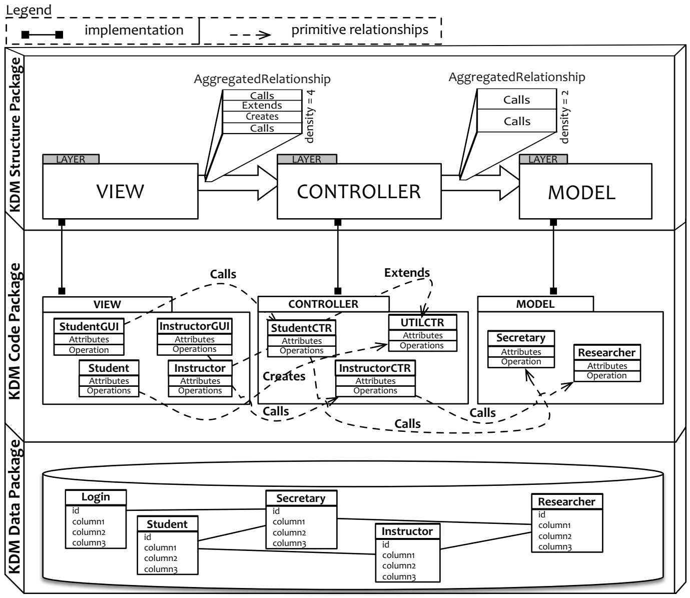

Fig. 1: System example.

JOS), 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 metaclass RelationalTable and its columns are mapped using the metaclasses UniqueKey and ColumnSet.

III. Motivation

Considering the system described earlier it is possible to identify two refactoring opportunities. The first one is related to the Student and Instructor classes, which are erroneously in the VIEW package and should be moved to the MODEL package using the *Move Classes* refactoring. The second one is regarding the attributes of the Student class that represent an address. A possible structural improvement could be turned these attributes into a new class Address - so the *Extract Class* can be applied here.

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.

In the matter of the refactoring *Move Class* (move Student and Instructor from VIEW 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 VIEW to the package CONTROLLER. 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.

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, 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.

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.

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 2.

In step [A], *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.

In the step [B], *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

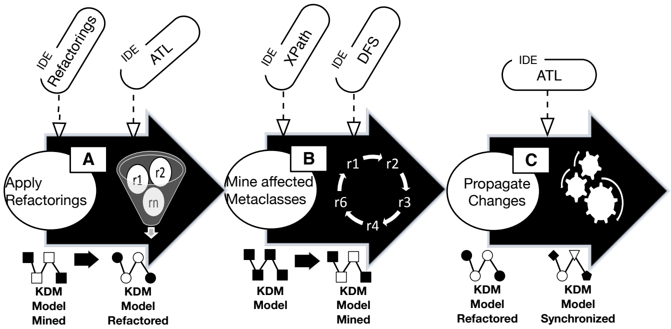

Fig. 2: Propagation-Aware Refactorings steps.

a set of queries that works on a KDM instance. In fact, this step uses depth-first search algorithm[1] to identify all affected metaclasses along with a set of queries.

In step [C], *Propagate Changes*, involves updating the elements identified in the step [B]. As in step [A], 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. Apply Refactoring

In the step [A] the engineer must apply a refactoring. A natural way of implementing refactoring in models is by means of *in-place transformations*[2]. 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). 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 3.

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

```
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. 3: 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 the refactoring's name is defined, *Move Classes*. 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), 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. 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), i.e., helpers in ATL are like methods/procedures in programing languages. 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 3, lines 13, 16, and 17 (`#parameter`). Therefore, before to apply the refactoring, the engineer should specify all the parameters. Considering our running example (depicted in Figure 1) the parameters would be: `Model`, `Student`, and `Instructor`, respectively. Afterward, this ATL is ready to be applied into a KDM instance. We have devised an Eclipse plugin to help the engineer to specify these parameters.

B. Mine Affected Metaclasses

The step [B] starts with our DI Algorithm that aims to identify all metaclasses and its relationships that use somehow the metaclass(es) that were refactored in step [A]. As input all the metaclasses that were used to apply an specific refactoring is needed. In addition, our DI Algorithm uses a set of queries that are performed on the KDM's instance to mine all the affected/linked metaclasses. All the queries were created using XPath.

Concerning the running example depicted in Figure 1 the engineer aimed to apply the refactoring *Move Class* - both

[1]From here on in Dependents Identification Algorithm (DI Algorithm)

[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.

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 which other metaclasses can be affected.

Therefore, firstly a query must be executed to get the root element in KDM. This query is represented as the first statement in Figure 4, 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 4 lines 2 to 5. The returned elements of these queries are used as input in our DI Algorithm.

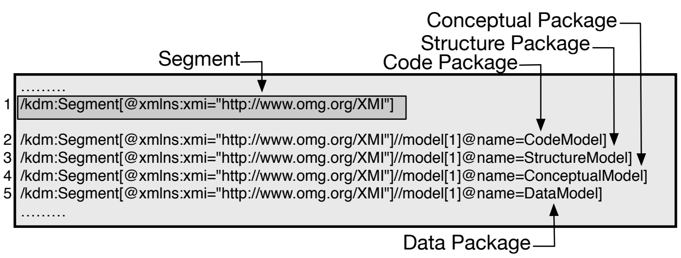

Fig. 4: Xpath used to return the KDM's root element, Segment.

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

```
begin
    foreach outgoing edge e = (u, v) of u do
        if vertex v as has not been visited then
            if vertex v contain implementation = true
            then
                foreach implementations element do
                    verify all elements in implementation
                end
                Mark vertex v as visited (via edge e).
                Recursively call DFS (G, v).
            end
        end
    end
end
```

Algorithm 1 depicts the DI 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 were refactored in Step [A] (e.g., for the refactoring *Move Class* three affected elements - `Student`, `Instructor`, and their package).

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 `implementation` without check, our algorithm should show a list of all affected metaclasses that is further used to propagated all changes throughout the KDM packages.

C. Propagate Changes

In this step, all propagations regarding an specific refactoring, e.g., *Move Class*, are implemented. Similarly to the step [A], this step is also defined by means of a set of ATL rules. Figure 5 shows an ATL used to propagate the changes regarding the refactoring *Move Class*. 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 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*.

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. Evaluation

This section describes the experiment used to evaluate the change propagation effectiveness of our approach. In fact, two experiment were conducted. The first experiment is called "Mining Study" and was planned to identify the effectiveness of our mining algorithm. Therefore, we have compared its result with an oracle in order to verify its correctness. The second experiment is referred as "Propagation Study" and was

```
....
rule propagationStructure {
  from
        source : MM1!Layer (source.allInstance
              -> select(e | e.refImmediateComposite = '#parameter'))
  to
        target : MM1!Layer (
         aggregated <- thisModule.getDensityAggregation(target.aggregated)
        )
}
rule propagationScenario {

  from
        source : MM5!RuleUnit (source.name = '#parameter')
  to
        target: MM5!ScenarioUnit (
         conceptualElement <-
              Sequence{thisModule.getRuleUnit('#parameter')...}
        )

}
rule propagationDataPackage {
  from
        source : MM!ClassUnit (source.allInstances()
              -> select(e | e.refImmediateComposite = '#parameter')
              ->collect(e | e.name = '#parameter' or e.name = '#parameter'))
  to
        target: MM4!RelationalTable
        (
              name <- source.name,
              itemUnit <- Sequence {columns} ->
                    union(source.codeElement->
                          select(e | e.oclIsTypeOf(MM!StorableUnit)))
        ),
        columns : MM4!ColumnSet (
              name <- '#parameter',
              type <- if (source.type.oclIsUndefined()) then
                          OclUndefined
              else
                          source.type->getType()
              endif
        )
}
....
```

Fig. 5: Chunk of code in ATL to perform the propagation after the application of refactoring *Move Class*.

planned to evaluated the correctness of the propagation given a set of refactorings. In addition, we have worked out two research questions, as follows:

RQ$_1$: Given some specific elements to be refactored, is the mining algorithm able to identify correctly all the dependent KDM elements?

RQ$_2$: Given a specific refactoring R, are all dependent elements identified in the oracle correctly refactored?

A. Goal Definiton

We use the organization proposed by the Goal/Question/-Metric (GQM) paradigm, it describes experimental goals in five parts, as follows: (*i*) **object of study:** the object of study is our approach; (*ii*) **purpose:** the purpose of this experiment is to evaluate the effectiveness of our approach, i.e, our mining affected metaclasses and the propagation of changes; (*iii*) **perspective:** this experiment is run from the standpoint of a researcher; (*iv*) **quality focus:** the primary effect under investigation is the precision and recall after applying the mining algorithm and a set of refactorings; (*v*) **context:** this experiment was carried out using Eclipse 4.3.2 on a 2.5 GHz Intel Core i5 with 8GB of physical memory running Mac OS X 10.9.2. The experiment can be defined as: **Analyze** the effectiveness of both the change propagating of our approach and the mining affected metaclasses, **for the purpose of** evaluation, **with respect to** precision and recall, **from the point of view of** the researcher, **in the context of** a subject program.

B. Hypothesis Formulation

In order to accomplish our goal, we explored the formalization of our research questions into the following hypotheses:

TABLE I: Hypotheses for the Mining Study

H$_0$	There is no difference in pattern recognition before and after to apply our mining affected metaclasses algorithm into the KDM model (measured in terms of the metric precision (P) and recall (R)) which can be formalized as: **H**$_0$**:** $\mu_{P_{Bf}} = \mu_{P_{Af}}$ **and** $\mu_{R_{Bf}} = \mu_{R_{Af}}$
H$_1$	There is a significant difference in pattern recognition before and after to apply our mining affected metaclasses algorithm into the KDM model (measured in terms of the metric precision (P) and recall (R)) which can be formalized as: **H**$_1$**:** $\mu_{P_{Bf}} \neq \mu_{P_{Af}}$ **and** $\mu_{R_{Bf}} \neq \mu_{R_{Af}}$

TABLE II: Hypotheses for the Propagation Study

H$_0$	There is no difference in propagation of changes before and after to apply a refactoring into the KDM model (measured in terms of the metric precision (P) and recall (R)) which can be formalized as: **H**$_0$**:** $\mu_{P_{Bf}} = \mu_{P_{Af}}$ **and** $\mu_{R_{Bf}} = \mu_{R_{Af}}$
H$_1$	There is a significant difference in propagation of changes before and after to apply a refactoring into the KDM model (measured in terms of the metric precision (P) and recall (R)) which can be formalized as: **H**$_1$**:** $\mu_{P_{Bf}} \neq \mu_{P_{Af}}$ **and** $\mu_{R_{Bf}} \neq \mu_{R_{Af}}$

There are two variables shown on each table: 'P' and 'R'. 'P' stands for Precision which is the ratio of the number of true positives retrieved/identified to the total number of irrelevant and relevant code elements retrieved/propagated. It is usually expressed as a percentage, see equation 1. R denotes Recall which is the ratio of the number of true positives retrieved/propagated to the total number of relevant code elements in the source code. It is usually expressed as a percentage, see equation 2.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \quad (1)$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (2)$$

C. Experiment Desing

For our evaluation, we need firstly transform a system into a KDM instance to apply our approach. However, due to the scarcity of complete KDM instances in the public domain, we adopted a reverse engineering approach and generated KDM instance from one system developed in Java by using MoDisco. This system is called LabSys (Laboratory System) and it is currently used by Federal University of Tocantins (UFT) for. It is used to control the use of laboratories in the entire university. LabSys was defined using the MVC architectural pattern. It contains a total of 15 packages, 113 classes, and 1307 methods. It is composed by three layers: `model`, `view`, and `controller`. Layer `model` owns the DTO (Data Transfer Objects) and DAOs (Data Access Objects), which is represented by `Data Package`. DTO represents domain entities such as laboratories, equipments, reservations, etc. DAO is the classes that performs the database access. Layer controller is responsible for the business rules that communicates directly with model layer. Finally, view layer is the part of the software system that performs direct interaction with the user and uses the resources of controller layer. In fact, we selected this system for our validation because its code have been devised by one of the authors of this paperdetected and analyzed manually

Currently, MoDisco only generates the KDM code package, other KDM packages are extremely important to evaluate our approach. Therefore, we have manually instantiated the followings KDM packages: Structure Package, Data Package, and Conceptual Package.

We selected three refactorings for our evaluation: *Extract Class*, *Move Class*, and *Pull up Method*. We applied each of the three refactorings to every possible location in KDM instance. It is worth to notice that all refactorings were applied completely automatically by means of our devised proof-of-concept tool. To deal with refactorings that go into infinite loops, we set three minutes timeout interval. More specifically, we applied the *Extract Class* to every class that had more than 300 LOC (Line of Code); we applied the *Move Class* to every class from a package to another package; we applied the *Pull up Method* to every method of a class that had a superclass that was not from a library, using every such superclass as the target of the pull-up. Then after applied all refactorings we counted whether them were successful, i.e., if the intended refactoring could be performed, and how many propagations were generated on the model. We also counted if our DI Algorithm was effectiveness to identify all affected metaclasses after applying a refactoring. We also measured both software metrics precision and recall after applying the refactoring on the KDM models.

D. Analysis of Data and Interpretation

The data of the first study is found on Table X

VI. Related Work

In [6], 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 [15] since it starts from the EMF metamodel applying a transformation rules.

In [16] 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 [17], 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.

The focus of our paper is the demonstration of propagation that must be performed in different static representations (views) of a given system. This means that we are not concerned with dynamic parts. As stated earlier, previous research has demonstrated concerns about the propagation of changes when modifications are made in models. However, the largest of them are concentrated in the propagation of changes between different metamodels. As KDM is a integrated model that can be seen as a set of metamodels, change propagation becomes trivial. This is because a certain model element is used in several places in general is referenced by its *id* without having to be duplicated in multiple locations. Some partial change propagations are already developed by MoDisco[3] plugin. For example, when a particular model element is removed, its *id* is removed from all the other places that it is used. This is considered a partial propagation, because it can, in most cases, inserting inconsistencies in the model.

Westfechtel *et al*. [10] presented a proposal that perform modification in static models, then all changes are propagated to behavioural models aimed at maintaining of after applying refactorings in models. Unlike these authors, this project aims to look more carefully for change propagation that need to be done when there are representations in different abstractions levels/view of the same lower-level element. It is not the purpose of our paper ensure that the refactorings maintain the observable behavioural of the system. A possible future work is to integrate the proposed of Westfechtel *et al*. [10] with our presented approach.

VII. Conclusions

Acknowledgments

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

[3]https://eclipse.org/MoDisco/

References

[1] 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

[2] 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.

[3] 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.

[4] 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.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Agosto 2000.

[6] 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.

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

[8] T. Mens, "On the use of graph transformations for model refactoring," pp. 219–257, 2006.

[9] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," *Software and Systems Modeling*, pp. 269–285, 2007.

[10] S. Winetzhammer and B. Westfechtel, "Propagating model refactorings to graph transformation rules," in *ICSOFT-PT 2014 - Proceedings of the 9th International Conference on Software Paradigm Trends, Vienna, Austria, 29-31 August, 2014*, 2014, pp. 17–28.

[11] 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.

[12] R. Durelli, D. Santibanez, B. Marinho, R. Honda, M. Delamaro, N. Anquetil, and V. de Camargo, "A mapping study on architecture-driven modernization," in *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*, 2014, pp. 577–584.

[13] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems," *Comput. Stand. Interfaces*, pp. 519–532, 2011.

[14] OMG, "Object Management Group (OMG) Architecture-Driven Modernisation," Disponível em: *http://www.omgwiki.org/admtf/doku.php?id=start*, 2012, (Acessado 2 de Agosto de 2012).

[15] 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.

[16] 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.

[17] 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.