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 [?]. 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) [?], [?], [?], 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 2010 (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 [?]. Behavior preservation in refactoring activities has received a lot of attention for years, both in source code and in models [?], [?], [?], [?]. 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 [?]. 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 [?], [?], [?], [?], [?], 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 [?], [?], 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 **??** a motivation is presented. Section III shown the proposed approach. In Section V, an empirical evaluation is presented. In Section VII there are related works and in Section VIII 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 [**?**]". 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 it allows modernization tools to exchange application metadata across applications, languages, platforms, and environments. This metamodel provides a comprehensive view of as-is application and data architectures, into a unique place; this is very different form conventional model-driven development techniques we have found on literature [**?**], [**?**]. 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 [**?**]. 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 - it supports a variety of languages, database structures, structures elements, middleware, Graphical User Interface, and platforms in a common metamodel.

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 are a set of relationship called Implementation, which are represented in Figure 1 by the symbol ■—■. The aim of this relationship 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

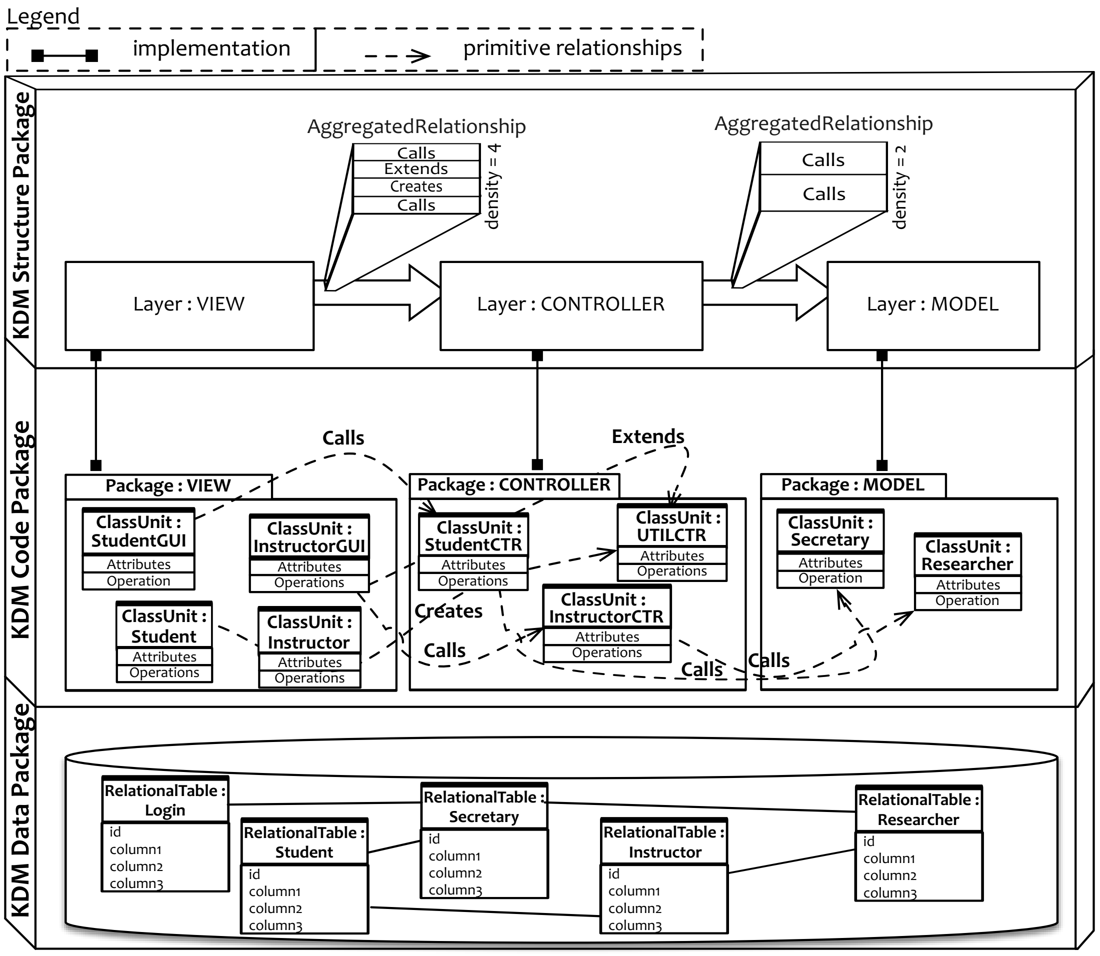


Fig. 1: System example.

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 (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 metaclass `RelationalTable` and its columns are mapped using the metaclasses `UniqueKey` and `ColumnSet`.

III. 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. Our approach is called Propagation-Aware Refactoring (PARef) and it is split into three steps, which are depicted by its corresponding letters and tittle in Figure 2.

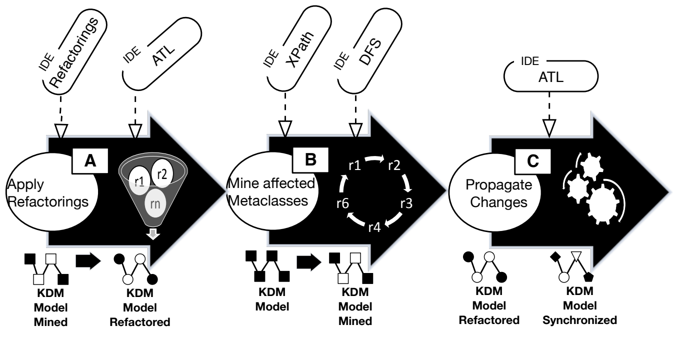


Fig. 2: Propagation-Aware Refactorings steps.

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 model-to-model (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/propagated 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 addition, 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/propagate 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 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.

```
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*.

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. For instance, he(she) should specify the source Package and two classes to be moved. 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.

It is important to highlight that this step was devised to be a decoupled module. In this way the modernizer could create his/her set of refactoring to KDM in ATL without worrying about the change propagation. The only task is to provide for our approach all necessary inputs to identify all affected metalclasses and to conduct the propagation of changes in others KDM levels.

B. Mine Affected Metaclasses

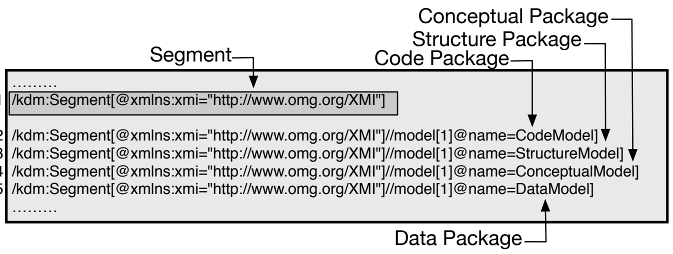
The step [B] starts with our DI Algorithm that aims to identify all metaclasses and its relationships that use somehow the metaclass(es) that were refactored in step [A]. As input all the metaclasses that were used to apply an specific refactoring is needed. For example, in the case of the refactoring *Move Class* it is necessary as input a package and a set of classes that were moved. Further, 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

[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.

using XPath.

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 as all the metaclasses that were used to apply the refactoring in Step [A].



```
.........
1 /kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]
2 /kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=CodeModel]
3 /kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=StructureModel]
4 /kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=ConceptualModel]
5 /kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=DataModel]
.........
```

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
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
```

Algorithm 1 depicts the DI Algorithm that is used to mine all the affected metaclasses. 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 reaches 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

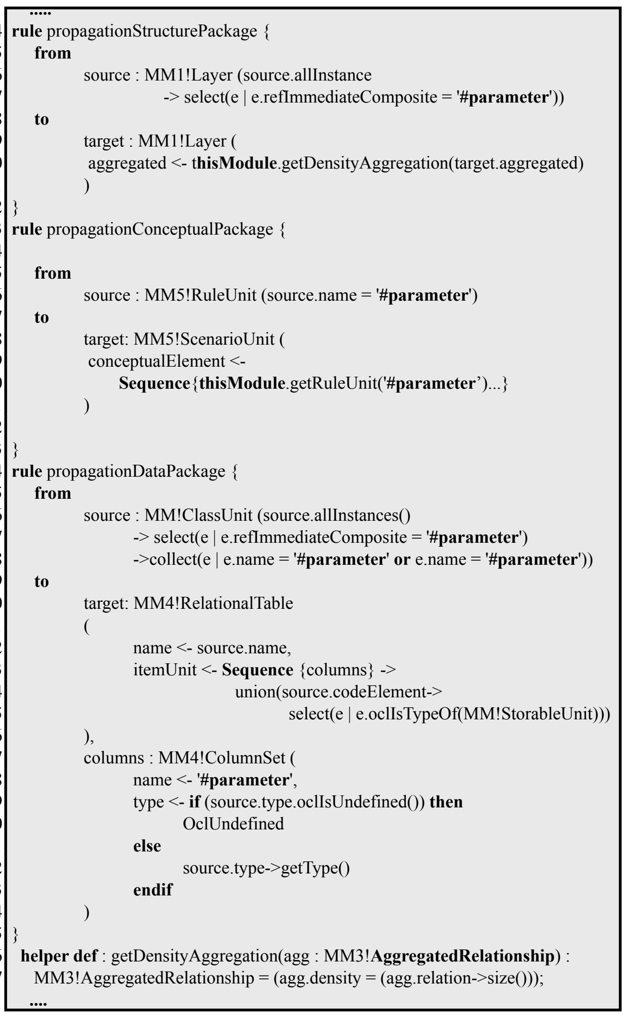
This step is a decoupled module that can be coupled to existing refactorings. In this way, existing users can write KDM refactorings in ATL without worrying about the change propagation. The only task is to provide for our component all the parameters it needs to conduct the propagation. In our approach these parameters are identified automatically in Step [B]. Similarly to the step [A], the step [C] is also defined by a set of generic ATL rules. Figure 5 shows a code snippet written in ATL that is used to propagate the changes. Due space limitation the whole ATL it is not 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 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 a refactoring, i.e, *Move Class*.

If the *Move Classes* refactoring is applied to transfer the class C1 to package P2, a natural propagation is to transfer the business rule B1 to another scenario. As defined in the second rule (lines 33 to 43) - this rule is used to propagate the changes throughout the `Conceptual Package`.

Finally, the rule defined in lines 44 - 65 aims to propagate the change to the `Data Package`. For each `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.

IV. Case Study

In this section we present a case study showing that our approach can be used to support the change propagation in KDM models. We have used a real-life legacy information system. Notice that the case study was carried out following the protocol for planning, conducting and reporting case studies proposed by Brereton et al. in [17] improving the rigor and validity of the study. The next subsections show more details about the main phase defined in this protocol, such as: background, design, case selection, case study procedure, data collection, analysis and interpretation and validity evaluation.



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

   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
            )
}
 helper def : getDensityAggregation(agg : MM3!AggregatedRelationship) :
   MM3!AggregatedRelationship = (agg.density = (agg.relation->size()));
 ....
```

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

A. Background

According to the protocol proposed by Brereton et al. in [**?**] firstly it is needed to identify previous research on the topic. Hence, in Section VII we stated some researches related to refactoring in models. However, none of them are using ADM/KDM. We are particularly focus on the propagation of changing in different views of a KDM model. In this context, the object of this study is the proposed approach, and the purpose of this study is the evaluation of the approach herein described related to its effectiveness and efficiency.

Therefore, taking into account the object and purpose of the study, it was defined one research question, as follows:

- $\mathbf{RQ}_1$: Given a set of refactoring, can the proposed approach propagate all the changes effectively throughout all KDM levels?

B. Design

The described case study consist of a single case [**?**]. It was focused on a single legacy system. To assess the effectiveness of the proposed approach through the $\mathbf{RQ}_1$, we use some oracles. As each refactoring has its own characteristics and modifies specific model elements, it is possible to predict all the expected changes in other KDM levels. So, considering our set of developed refactorings, we had to develop some oracles for each refactoring. The complete oracle can be see at www.mudar.com.br.

C. Case Selection

In this section is described the suitable case that was chosen to be studied. Some criteria were applied to select the suitable case, as follows: (i) it must be an enterprise system, (ii) it must be a Java-based system, (iii) it must be a legacy system and (iv) it must be of a size not less than 10 KLOC. After applying these criteria we chose LabSys (Laboratory System) it is currently used by Federal University of Tocantins (UFT). It is used to control the use of laboratories in the entire university.

D. Case Study Procedure

In this section is shown how the execution of the study was planned. Notice that the execution was aided by the tool developed to support the proposed approach. The case study was carried out in a machine with an Intel Core I5 CPU 2.5GHz, 8GB of physical memory running Mac OS X 10.8.4.

The proposed approach uses as initial artifact a KDM instance,. Therefore, firstly we adopted a reverse engineering to transform the LabSy source-code into a KDM instance to apply our approach. In this step we have used MoDisco [**?**], which is a parser that get as input java source-code and then return as output a KDM instance. 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`. After applying LabSys to MoDisco we gathered a KDM instance that contains 29,444 number of model elements (in this context KDM objects in the model) and the memory used on hard drive disk after XMI serialization is 7.639 MB.

Furthermore, to perform the case study we selected four refactorings: *Extract Class*, *Move Class*, *Extract Layer* and *Remove Class*. 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 one class that had more than 300 LOC (Line of Code); we applied the *Move Class* to a set of class from a package to another package; we applied the *Extract Layer* to a layer that contains at least 20 classes; finally we applied the *Remove Class* to a class that contained at least 15 primitive relationships. After applied all refactorings we verify whether them were successful, i.e., if the intended refactoring could be performed, and if all the expected propagations were generated on the model.

E. Data Collection and Interpretation

We verify, based on a set of oracle, whether all refactoring were successfully propagated throughout all KDM models. By using these information gathered we can draw conclusion and answer the **RQ**$_1$.

Table I summaries the results related to each refactoring applied and its respective propagations. "P.C?" stands for "Propagation Corrected?". These tables show the propagation regarding to the followings KDM packages: `Structure Package`, `Data Package`, and `Conceptual Package`. As can be seen all the changes were effectively propagated throughout all KDM levels. Which means that in this case our approach could automatically execute truly relevant propagation throughout all KDM levels when dealing with the refactorings: *Extract Class*, *Move Class*, *Extract Layer* and *Remove Class*.

Globally, the use of our approach in the context of this refactoring case study has been a success. As a result of the process, approximately 60 K SLOC from the whole application were concerned and so automatically refactored. An effective performance gain and overall readability improvement of the modified KDM model parts have then been confirmed. Thereby, the **RQ**$_1$ can be answered as true, that is, the proposed approach can propagate changes effectively throughout all KDM levels.

V. Evaluation

This section describes the experiment used to evaluate the effectiveness of our mining algorithm, step [B] of our approach. Therefore, we have compared its result with an oracle in order to verify its correctness. In addition, we have worked out one research question, as follows:

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

A. Goal Definiton

We use the organization proposed by the Goal/Question/-Metric (GQM) paradigm, it describes experimental goals in five parts, as follows:

- **object of study:** the object of study is our approach;
- **purpose:** the purpose of this experiment is to evaluate the effectiveness of our mining approach;
- **perspective:** this experiment is run from the standpoint of a researcher;
- **quality focus:** the primary effect under investigation is the precision and recall after applying the mining algorithm;
- **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.

Our experiment can be summarized using Wohlin et al.'s template [**?**] as follows:

Analyze the effectiveness of our mining algorithm
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. Effectiveness Analysis

Herein we present an effectiveness analysis aiming to determine the recall and precision of our approach. To do that, we have applied our mining approach, step [B], in one system and compared the results with oracles and an own manual analysis.

The system we have used as case studies was LabSys, the same system used in the case study. This analysis employs the metrics Recall and Precision, which are described below:

- Precision is the ratio of the number of true positives retrieved to the total number of irrelevant and relevant KDM elements retrieved/propagated. It is usually expressed as a percentage, see equation 1.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{1}$$

- Recall is the ratio of the number of true positives retrieved to the total number of relevant KDM elements in the KDM instance. It is usually expressed as a percentage, see equation 2.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{2}$$

C. Experiment Desing

For our evaluation, we also used the same system described in Section IV, LabSys. As stated before, firstly we transformed it into a KDM instance to apply our approach by means of MoDisco. Furthermore, to evaluate our mining approach it was necessary to choose some refactoring. As a matter of fact, it is important to know its parameters once our mining algorithm uses them to identify all affected metaclasses. Therefore, we selected four refactorings and use its parameters as starting point of our mining approach. The chosen refactorings were: *Extract Class*, *Extract Layer*, *Move Class*, and *Remove Class*. Then after applied our mining approach we counted whether all the affected metaclasses were successful identified.We also measured both software metrics precision and recall after applying the mining algorithm on the KDM models.

D. Analysis of Data and Interpretation

Table II presents both metrics: precision and recall. The each column represents the effectiveness analysis, whose goal is to analyze the recall and precision of our mining approach. In order to calculate the precision and recall values we used an oracle as the base for the comparison. This oracle has been build based on our experience in KDM models. The process of checking and calculating these metrics were very time consuming because we needed to compare the log produced by our mining approach with the xml files (KDM instance). In order to help us in the identification of the most significant precision and recall we have built a bar-plot that can be seen in Figure 6

TABLE I: Propagations for the refactorings: Extract Class, Extract Layer, Move Class, and Remove Class

Refactoring	**Extract Class**	**P.C?**
Code	Create an instance of ClassUnit that represent the new Class	Yes
	Move all StorableUnits to the new ClassUnit	Yes
	Move all MethodUnit to the new ClassUnit	Yes
	Create an intance of HasType, which represent an association between the new ClassUnit and the old ClassUnit	Yes
Structure	N. A	N.A
Data	Create a instance of RelationalTable owning the name of the new ClassUnit	Yes
	For each StorableUnit it is necessary to create a ItemUnit, which represent the RelationaTable columns.	Yes
	Create an instance of UniqueKey that represent the primary key of the RelationalTable.	Yes
Conceptual	N. A	N. A

Refactoring	**Extract Layer**	**P.C?**
Code	Create an instance of Package	Yes
	Move a the selected ClassUnit from a Package to the new Package	Yes
Structure	Create an instance of Layer	Yes
	Create an instance of AggregationRelationship between the new Layer and the old one	Yes
	Associate the new Layer by means of the association implementation with the new Package	Yes
	Summing up all primitive relationship to compute the meta-attribute density	Yes
Data	N. A	N. A
Conceptual	If the moved classes are associated to any conceptual elements by means of the association implementation these conceptual elements should be moved to a correspondent associated element of the target Package.	Yes

Refactoring	**Move Class**	**P.C?**
Code	Move an specific ClassUnti from a source Package to a target Package	Yes
Structure	Package If the target Package is associated to an architectural elements by means of the association implementation the value of meta-attribute named density should be propagated	Yes
Data	N. A	N. A
Conceptual	If the moved class is associated to any conceptual elements by means of the association implementation this conceptual elements should be moved to a correspondent associated element of the target Package.	Yes

Refactoring	**Remove Class**	**P.C?**
Code	Delete the selected instance of a ClassUnit	Yes
Structure	If the removed ClassUnit was contained into a specific Structure element then summing up all primitive relationship and overwrite the meta-attribute density	Yes
Data	if the removed ClassUnit was associated with an instance of RelationalTable, then it should also be removed	Yes
Conceptual	if the removed ClassUnit is associated to any conceptual elements by means of the association implementation these conceptual elements should be removed	Yes

TABLE II: Values of precision and recall.

	Efectiveness Analysis							
	Extract Class		Extract Layer		Move Class		Remove Class	
System	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
LabSys	100%	100%	80%	100%	100%	95.11%	100%	90.3%

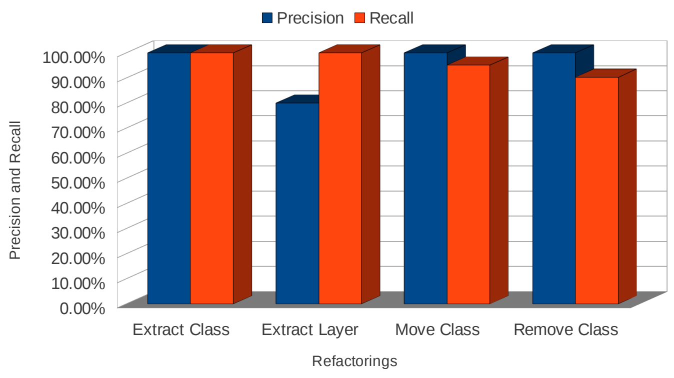


Fig. 6: Bar-plot for precision and recall of each refactoring.

Observing both Table II and Figure 6, it is possible to see that for the refactoring *Extract Class*'s parameters we got 100% of precision and recall; that is there are no false negatives or positives. However, notice we got 80% of precision for the refactoring *Extract Layer*'s parameters. This happened because our mining algorithm has recognized more similar metaclasses in *Extract Layer* than in *Extract Class* increasing the number of false positives, as most of these metaclasses had not relation with the parameters.

Obviously, our mining algorithm failed in some cases because although some metaclasses are similar, the semantic is completely different. For example, we could have two similar instance of an specific metaclass, therefore the algorithm would identify just one. As can be seen in Table II it is clear that our mining affected metaclasses algorithm helps to find metaclasses which are related with a particular refactoring but it is not foolproof. Nevertheless, empirically we can say that the algorithm add value to the whole solution.

As can be seen in our analyses, good recall and precision values were obtained using our mining affected metaclasses algorithm. Therefore, this can enable other groups to proceed researching on data mining techniques. Clearly, we cannot guarantee the same level of recall and precision but maybe it is possible to keep improving these metrics by using other data mining techniques.

E. Threats to Validity

The lack of representativeness of the subject programs may pose a threat to external validity. We argue that this is a problem that all software engineering research, since we have theory to tell us how to form a representative sample of software. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings. A threat to construct validity stems from possible faults in the implementations of the techniques. With regard to our mining techniques, we mitigated this threat by running a carefully designed test set against a complex system.

VI. A Brief Discussion

Notice that some propagations can also be considered as refactoring and vice versa. What characterize them is how they are used in a specific moment and not the implementation by itself. This is like having a set of refactoring that anyone can trigger anyone. When this is the case, the modernization engineer can directly apply both, unlikely propagations which clearly cannot be directly applied from the user, as it is shown in [**?**]. This is generally the case for moving refactorings, as the moving of an element from a container to another is independent of both the container and their abstraction level. For example, suppose the existence of a class C1 belonging to a package P1. Consider also that C1 is the implementation of a business rule B1 which is inside a scenario S1 and that the package P1 is the implementation of the Scenario S1. So, C1 = B1 and P1 = S1. If the *Move Class* refactoring is applied to transfer the class C1 to package P2, a natural propagation is to transfer the business rule B1 to another scenario. However, if the modernization engineer is using a modeling environment which provides a business rule view, (s)he could also have available for him(her) a moving business rule refactoring. In this case, the natural propagation would be to transfer the corresponding classes from one package to another. Therefore, we can see that in some cases there is bidirectional flow, which can be started from any point.

The most important thing about this discussion is that this categorization lead us to make good designs in terms of refactorings and propagations. That is, for refactorings that fall in this category, it is very important to implement them as separated and decoupled modules which can be called directly from the user. So, all of our refactorings were implemented like that.

VII. Related Work

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

In [**?**] 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 [**?**], 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.* [**?**] 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.* [**?**] with our presented approach.

VIII. Conclusions and Future Work

Acknowledgments

Rafael Serapilha Durelli would like to thank the financial support provided by FAPESP, process number 2012/05168-4. Fernando Chagas and Bruno Santos would also like to thank CNPQ.

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 &#8211; 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]https://eclipse.org/MoDisco/

[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] P. Brereton, B. Kitchenham, D. Budgen, and Z. Li, "Using a protocol template for case study planning," in *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. Swinton, UK, UK: British Computer Society, 2008, pp. 41–48.

[16] H. Bruneliere, J. Cabot, G. Dupe, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, pp. 1012 – 1032, 2014.

[17] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[18] 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.

[19] 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.

[20] 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.