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 desynchronized. In this paper we present a tool-supported KDM-specific approach for updating dependent models when specific elements are refactored. Our approach involve two main steps; the identification of all changed KDM model elements (dependent on the refactored ones) and the application of changes (propagation) in order to keep all the models synchronized. We have conducted two evaluation that shows our approach reached good accuracy and completeness levels.**

I. INTRODUCTION

Architecture-Driven Modernization (ADM) advocates the conduction of reengineering processes following the principles of Model Driven Architecture (MDA) [1]–[4], i.e., all software artifacts considered along the process are models. A typical ADM-based modernization starts with a reverse engineering phase to recuperate a model representation of the legacy system; proceeds by applying refactorings over the recuperated model and finalize by generating the modernized system.

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 system and retain the dependencies among them. This third point is possible thanks to several KDM metamodels/packages that are focused on specific views or abstraction levels. Examples are: source-code (Code metamodel), behaviors (Action metamodel), architecture (Structure metamodel), business rules (Conceptual metamodel), data (Data metamodel), events (Event metamodel), GUI (UI metamodel) and deployment (platform metamodel).

Unlikely existing metamodels, KDM aims at putting together all the views of the system into a unique place, so, it can be considered as family of metamodels, since all of them share a consistent and homogeneously terminology (metamodel sintax) [5].

It is well known that refactoring activities are central to any modernization process. Refactorings are defined as the process of modifying the internal structure of software without changing its external observable behavior [6]. When a system is represented by using several different models, a common accident that arises during refactorings is to desynchronize the models, ending up with inconsistency models after the refactoring. To solve this problem, an alternative is to apply a "change propagation" technique, whose goal is to identify and update all the model elements dependent on the refactored element [7]–[10].

Change propagation can be classified in two types: dynamic (behavior) and static. Dynamic Change Propagation has been the main focus for decades, trying to preserve the system behavior when refactoring a static model. On the other hand, Static Change Propagation, which is the focus of our work, has only more recently received attention, but it is also possible to find a number of research on this line of thought [7]–[10]. The goal of this type of propagation is updating a static model after refactoring another static model.

Most of the existing research on Static Propagation concentrate on propagating changes in different and external models, for example, when changing an UML class model, changes are propagated to a data model, usually from another vendor [11]. Besides, most of the change propagation solutions are specific to proprietary models, preventing or making difficult their application in other models, like KDM [7], [10]. To the best of our knowledge, up to this moment, there is no research concentrated on investigating change propagation in KDM.

In this paper we present a tool-supported approach for propagating changes when refactoring KDM models. Employing our approach, modernization engineers can concentrate just on the development of the refactorings, without worrying about the change propagation, which is a time-consuming and error-prone task. Our approach is supported by an Eclipse plug-in that allows model transformations languages (refactorings that are written in ATL, QVT, ETL, etc) be executed while the propagations are conducted in a transparent way.

The workflow of our approach is the following: firstly the modernization engineer either creates or reuses an specific KDM refactoring, this refactoring can be defined by means of a set of model transformations languages such as: ATL, QVT, ETL, etc. Then the engineer apply such refactoring in a KDM instance. Then after that, our approach performs its three steps. The first step performs a diff between the refactored KDM instance with the original KDM instance (the instance before engineer applies any KDM refactoring). As output this step provides a list that contains all KDM elements involved in the KDM refactoring. After that, the second step is performed, which automatically uses the list provided by the earlier step as input to our Java mining algorithm. This algorithm aims to identify all KDM model elements that were modified by a refactoring, this step also uses a set of queries defined using JPath to assist the mining algorithm. Similarly to the first step, the second one also provides a list as output that contains all KDM models elements that were affected/modified by a refactoring. Finally, the third step automatically uses the output from the last step along with a set of ATLs (transformations rules) predefined to realize the change propagation throughout all KDM levels. We have implemented the approach in a generic way as a decoupled module, which can be coupled to existing refactorings.

We have carried out an experiment and a case study in order to evaluate our two-step approach. The former was performed to evaluate the first step of our approach, i.e., analyze the effectiveness of our Java mining algorithm for with respect to precision and recall. The latter was carried out to evaluate the effectiveness and efficiency of our ATL script (change propagation), i.e., the case study aimed to verify if all propagations were performed corrected after to apply a set of refactorings.

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 IV shown the proposed approach. In Section VI, an empirical evaluation is presented. In Section VII there are related works and in Section IX there are the conclusions.

II. ADM and KDM

ADM is the process of understanding and evolving existing software systems taking model-driven principles into account [1]. A typical ADM process involves three main phases: Reverse Engineering, Refactorings/Optimizations and Forward Engineering. In reverse engineering, 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.

KDM is the most important meta-model of ADM (ISO/IEC 19506), providing a comprehensive view of as-is application and data architectures, into a unique meta-model. This is different from conventional model-driven development techniques we have found on literature [3], since many of them employs several meta-models, from different vendors, along the process [12]. KDM can be seen as a family of meta-models, as it contains twelve packages; each one representing a meta-model that concentrates on a different view of the system. Thus, by using its meta-models, it is possible to have a number of views of a system. For example, it is possible to have a low level system representation, describing source-code details and several others views of the system, such as an architectural view, a data view, a business rule view, a behavioral view, etc. Moreover, as KDM groups a set of meta-models, all of them share the same terminology, i.e., all of the meta-models know the main meta-model elements, such as ClassUnit, KDMEntity and MethodUnit, etc.

Considering the scope of this paper, some important KDM packages are Code, Structure, Data and Conceptual. Code package provides a lot of meta-classes for representing source code details, such as MethodUnit (methods), ClassUnit (classes) and StorableUnit (attributes). Structure package is devoted to represent the architecture of the system, employing architectural concepts commonly find in the literature. So, it offers meta-classes for representing layers (Layer meta-class), subsystems (Subsystem meta-class), components (Component meta-class) and architecture views (ArchitectureView meta-class). It also offers a special kind of relationship called Aggregated (AggregatedRelationhip meta-class), whose goal is to relate architectural elements with each other. An important characteristic of this relationship is that it acts as a container of primitive relationships, i.e., it is possible to group several primitive relationships within it.

Data package aims at representing the database structure of the system, providing meta-classes for representing tables and their attributes (Columns, primary key, etc). Conceptual package offers meta-classes for representing conceptual elements of a system, such as business rules (RuleUnit meta-class), scenarios (Scenario meta-class), etc.

The main goal is to allow a complete representation of systems, ranging from low to high-level views. All of the aforementioned meta-models, although being in different abstraction levels, can be interrelated to each other. For example, consider the existence of a Java package P1 (Package meta-class) that contains a class C1 (ClassUnit meta-class). This package can be the source-code realization of a Layer L1 (Layer meta-class) and, at the same time, the realization of a Scenario S1. The class C1 can also be the realization of a business rule B1 (RuleUnit meta-class) that is inside the scenario S1.

III. A Running Example

Figure 1 presents an example that is used throughout this paper. It is shown schematically how KDM can be used for representing three views/abstractions 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 of the three big rectangles represents an instance of an internal meta-model. So, there is an instance of the Code

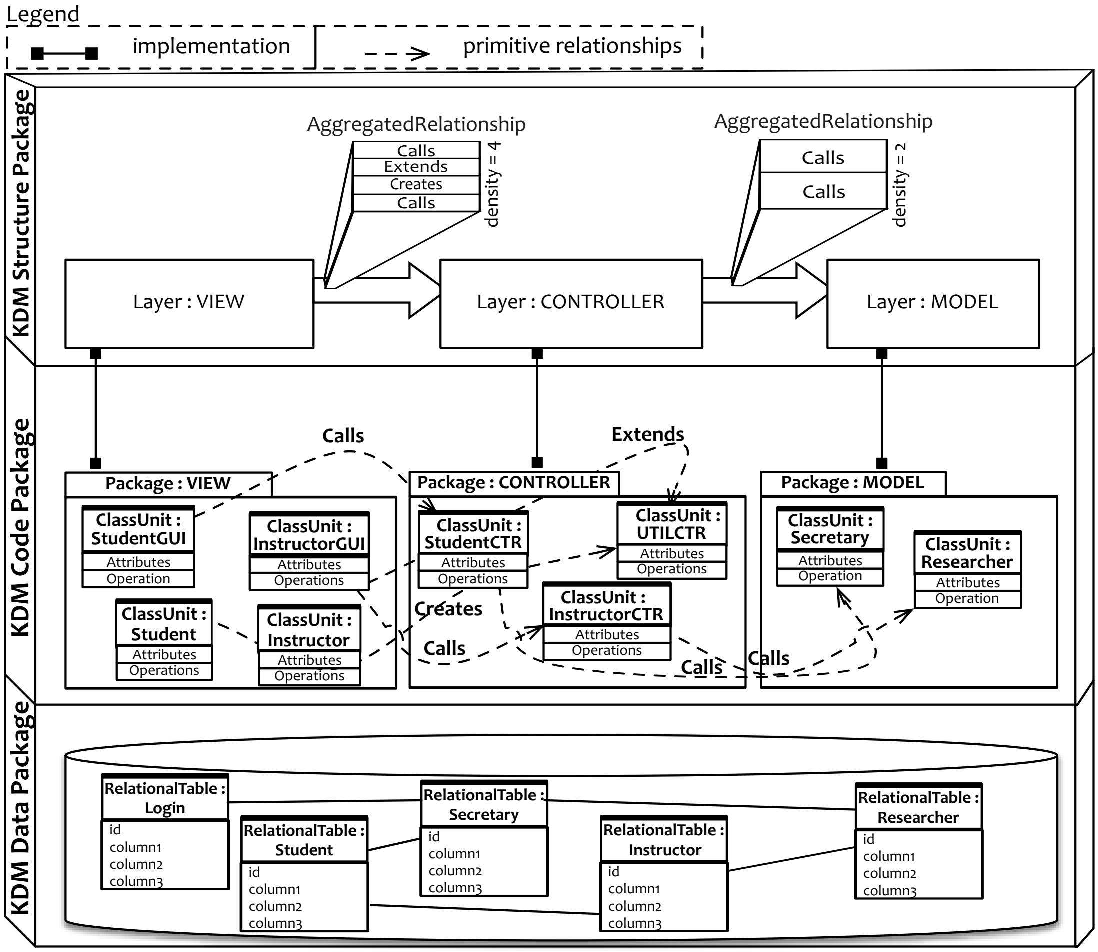

Fig. 1: System example.

meta-model (middle), another instance of the Structure meta-model (upper part) and another one of the Data meta-model (lower part).

Besides, each of the smaller internal elements (classes, packages, layers, relationships, etc) contains also instances of KDM meta-classes. Notice we are using the pattern **instance name: Meta-class name** in the name of every element so that the name of the meta-class can be seen. As this is an MVC-based system, Code View contains three instances of the Package meta-class: VIEW, CONTROLLER and MODEL. Each of them involving some ClassUnit instances. The classes are related to each other by means of static (associations, inheritance, interface realizations and imports) or dynamic (calls, object creation, parameter passing, etc) relationships. Each of these relationships is also an instance of specific meta-classes.

The Architecture View represents the system architecture. In this example each smaller rectangle represents an instance of a meta-class called Layer. The system is organized in three layers: 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 intention of this type of relationship is to denote that a specific higher-level abstraction is realized by one or more lower level code elements. In this example, 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. The `Implementation` relationship is very important in this work, as it is the main link between meta-models in different abstraction levels. Observe in the figure that the unique route between the views are by means of these relationships.

KDM also possesses an important relationship type called AggregatedRelationship whose aim is to represent dependencies between architectural elements of the Structure Package. Using this relationship it is possible to put together several primitive relationships in a unique “channel”. For example, there are thicker relationships between the layers representing the AggregatedRelationship. It is also possible to see that these relationships aggregate some primitive relationships. For example, 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 is called density, that represents the amount of primitive relationships exists inside it.

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 meta-class RelationalTable and its columns are mapped using the meta-classes UniqueKey and ColumnSet.

An evident problem in this example is the presence of the classes `Student` and `Instructor` in the VIEW package. A possible solution to this problem is to apply the *Move Class* KDM refactoring as presented in Figure 2.

```
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                                              parameters
11  rule moveClass {
12     from
13       source : MM!Package (source.name = 'MODEL')
14     to
15       target: MM!Package (
16         codeElement <- Sequence{thisModule.moveClass('Student', 'Instructor')
17     )
18  }
19
20  helper def : moveClass (className : Sequence(String)) :
21    MM!ClassUnit = MM!ClassUnit.allInstances()->any(e | e.name = className.name);
```

Fig. 2: Chunk of code in ATL to perform the refactoring *Move Class*.

Lines 14 though 19 the *Move Class* is actually defined. Lines 21 and 22 there is a helper, in ATL helpers are like methods in programing languages. This helper is used to verify if the ClassUnit is the correct class to be moved. Almost all refactorings need some input parameters that should be properly informed by the user. For instance, in the code depicted in Figure 2, lines 13, 16, and 17 the software modernizer informed three parameters (see ❶), i.e, he(she) specified the source Package and two ClassUnits that he(she) would like to move. Afterward, this ATL is ready to be applied into a KDM instance.

Observing the refactoring defined in Figure 2 it is fairly evident that both classes `Student` and `Instructor` should be moved to the MODEL package. However, moving these classes will turn the KDM instance inconsistent, because the "density" value will turn wrong, i.e., AggregationRelationShip between the layer VIEW and the layer CONTROLLER would change from 4 to 2 - once the primitives relationships Creates and Extends would no longer exist from the package VIEW to the package CONTROLLER. In the same way, the result of *Move Class* refactoring should also 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`.

These propagation seen to be easy to apply, however, in a complex system comprising all kdm's packages/levels propagate all changes after a refactoring is a difficult and error prone task. Even identifying the affected parts of the KDM's packages/levels is not an easy and straightforward process. A typical refactoring written in ATL is shown in Figure 2.

IV. Propagation-Aware Refactorings

In this section our approach named Propagation-Aware Refactoring (PARef) is presented. PARef aims to propagate changes, in a cascade way, throughout all the KDM's levels during a refactoring. The intention is to keep the consistency/synchronization among all the KDM's views during a refactoring activities.

Figure 3 shows the workflow of our approach. As noted it contains three steps, [A], [B], and [C] all contained into the gray cube. Outside of our approach there is the Refactoring activity, see the white rectangle. This activity is a normal and conventional model refactoring activity - this activity is out of scope of our approach is responsibility of the software modernizer to develop or reuse any model refactorings (in ATL, ETL, QVT) and apply them into a KDM model. After that, the first step, [A] is trigged then a diff between the refactored KDM instance and the original KDM instance (the instance before one applies a KDM refactoring). The output of this diff is a list that contains all KDM model elements involved during the KDM refactoring. From this point onward, the step [B], called *Identifying Points to Propagate* aims to gather all the KDM elements that need to be updated/synchronized as a result of a refactoring. This step runs a depth-first search algorithm[1]. This algorithm uses as input the list that was obtained from the diff between the refactored KDM and original KDM instance (step [A]). Then, It also uses the refactored KDM instance to generate as output all meta-classes that possesses dependencies with the elements to be refactored. The third step [C], called *Performing Propagation*, objectives to effectively perform in a cascade way all the changes/updates in the KDM model. The input for this steps are the elements to be changed (provided by the step [B]) and the output is the KDM model updated/synchronized.

The step [A], is technically supported by the framework EMF Compare once it provides comparison and merge facility for any kind of model. EMF Compare was reused and extended to compare instances of KDM models. The step [B], is technically supported by an *Identification Engine* whose the core is a depth-first search algorithm along with a set of queries that are performed over a KDM model.

The step [C] is technically supported by a *Propagation Engine*, whose core is a set of pre-defined transformation rules devised with ATL that works in cascade way. Herein all the transformation rules act as a chain of transformations that are executed together in order to update/propagate all the changes throughout KDM's views. More details on each step are provided in the next sections.

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

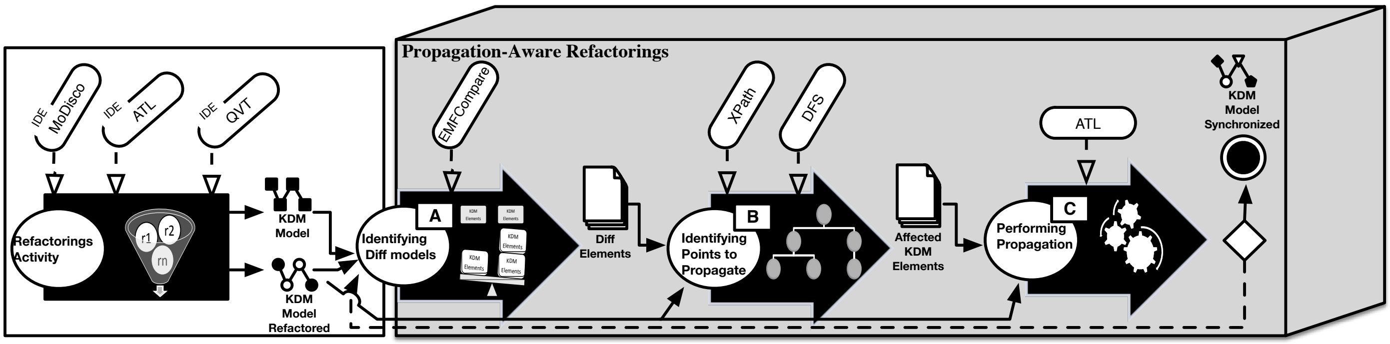

Fig. 3: Propagation-Aware Refactorings steps.

A. Identifying Diff Between Models

The problem of model diff is intrinsically complex. For instance, if a ClassUnit C is deleted, its transitive parts and attached associations are typically also deleted. So, computing the difference results in a large number of detail changes that might be complex to implement. Therefore, in this step we have used the EMF Compare[2], which is a framework that can easily reuse and extend to compare instances of any models, in our case KDM models. We have chosen to use EMF Compare because it was designed with scalability in mind in order to support comparisons of large fragmented models.

To start this step, our modified EMF Compare needs two instance of KDM as input: i) the refactored one (left side), and ii) the original (right side). The EMF Compare analyses, whether the KDM elements are equal or if they present differences (for example, the name of the class has been changed from Class1 to ClassX, or a class has been moved from a Package to another Package, etc). EMF Compare iterates over all of our KDM elements, be they unmatched (only one side has this object), couples (two of the three sides contain this object) and compute any difference that may appear between the sides. For example, a KDM element that is only on one side of the comparison is a KDM element that has been added, or deleted. But a couple might also represent a deletion: during three way comparisons, if we have an KDM element in the common ancestor (origin) and in the left side, but not in the right side, then it has been deleted from the right side version. The output of this step is a list that contains all KDM model elements involved during the KDM refactoring.

B. Identifying Points to Propagate

The step [B] starts with DI Algorithm to identify all affected meta-classes along with a set of queries. The DI Algorithm that aims to identify all meta-classes and its relationships that use somehow the meta-class(es) that were refactored. As input the list obtained from the EMF Compare in Step [A] is needed. For example, in the case of the refactoring *Move Class* the list would contains 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 meta-classes. All the queries were created using XPath. We have decided to use XPath because it is a well-know and well-documented language.

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 meta-class `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 the list obtained from the Step [A].

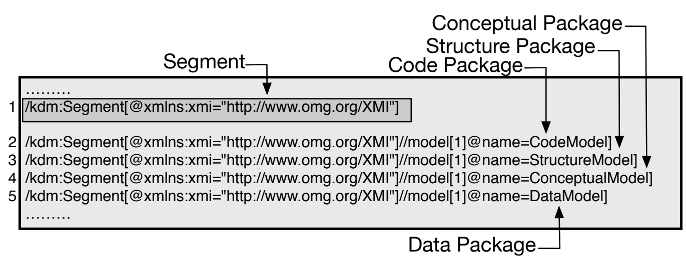

Fig. 4: Xpath used to return the KDM's root element, Segment.

Algorithm 1 depicts the DI Algorithm that is used to mine all the affected meta-classes. More specifically, the algorithm works as follows: first it is necessary to pick a starting point, i.e., the meta-class `Segment`. Visit the `Segment`, push it onto a stack, and mark it as visited. Then it is necessary to go to the next meta-class 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 meta-class. Then the algorithm checks to see if the `Segment` has any unvisited adjacent meta-class. If it does not, then it is necessary to pop it off the stack and check the next meta-class. If the algorithm finds one (unvisited meta-class), it starts visiting adjacent meta-classes until there are no more, check for more unvisited adjacent meta-classes, and continue the process always verifying the association named `implementation`. When the algorithm finally reach the last meta-class on the stack and there are no more adjacent, unvisited meta-classes that

[2]https://www.eclipse.org/emf/compare/

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 meta-class, i.e., Segment, and eL is a
       set of elements to verify
Output: A collection of affected meta-classes
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
```

contains the association `implementation` without check, our algorithm should create a list of all affected meta-classes that is further used to propagated all changes throughout the KDM levels.

C. Performing Propagation

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, which is a time-consuming and error-prone task. 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] and are used in Step [C]. Similarly to the step [A], where the modernizer has to define a set of model transformations rules to perform the model refactoring, here a set of generic and pre-established model transformations (written in ATL) are used. The difference is that in Step [A], the modernizer can either create or reuse a KDM refactoring, otherwise in Step [C] all rules were beforehand defined to perform the propagation of changes (in a cascade way) after the application of a KDM refactoring. In addition, these ATL rules (the propagations) require a set of *mininum* parameters that should be informed before realize all the propagations. As already mentioned these parameters are the output from Step [B], which is a list containing all KDM affected elements.

In order to bound these parameters along with the output of Step [B] our approach performs a static analysis (parsing) of all generic ATL rules and identifies places that must be replaced in with the Step [B]'s output (KDM affected elements), i.e, all the places where parameters are needed. This is particularly necessary in our approach because ATL does not enforce type correctness, hence rules written in ATL may be ill-typed. Moreover, the creation of a suitable propagation requires precise parameters (meta-classes) informations. It is important to highlight that this static analysis is done totally automatically and transparently by means of our Eclipse plug-in. The aim is having a decouple module of KDM propagation as simple as possible to facilitate the integration with any refactoring defined in ATL in the context of KDM model and also to promote the reuse. Therefore, the software modernizer does not have to worry about devising the propagation of changes, which usually is harder than just the creating of a KDM refactoring. In addition, if the static analysis detect errors, the software modernizer is required to fix and inform the correct parameters, otherwise, all changes are propagated in all KDM levels automatically/transparently

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. Note that all strings, **'#parameter'**, are changed during the static analysis along with the step [B]'s output. 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` (meta-class that represent all the attributes that a class holds) as well as its types.

Although we have used a simple refactoring as example, by observing both Figure 2 and Figure 5 one can notice that the refactoring itself usually is less complex/verbose to devise than the Propagation Engine, i.e., a set of rules defined to propagate the changes in KDM levels tend to be more complex/verbose than KDM refactorings. Therefore, providing a module that can be plugged over existing KDM refactorings in order to propagate the changes can assist the software modernizer.

V. 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 [13] improving the rigor and validity of the study. The next subsections show more details about the main phase defined in this protocol.

```
.....
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*

Firstly it is necessary to identify previous research on the topic [13]. Hence, in Section VII we stated some researches related to propagation of changes in models. To the best of our knowledge, up to this moment, there is no research concentrated on investigating change propagation in KDM. We are particularly focus on the propagation of changing in different views of a KDM model. The object of this study is the proposed propagation changes approach (PARef), and the purpose of this study is to evaluate the effectiveness and efficiency of our approach. Taking into account the object and purpose of the study, it was defined one research question, as follows:

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

B. *Design*

The described case study consist of a single case [13]. It was focused on a single legacy system. To assess the effectiveness of the proposed approach through the **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 seen at www.mudar.com.br.

C. *Case Selection*

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 (Kilo of Lines of Code). After applying these criteria we have chosen LabSys (Laboratory System) that is currently used by Federal University of Tocantins (UFT) 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 an Eclipse plug-in that we 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 [14], which is a framework 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 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 meta-classes in the model) and the memory used on hard drive disk after XMI serialization is 7.639 MB.

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 classes 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 successfully propagate throughout KDM levels, i.e., if the intended refactoring could be performed and if all the expected propagations were generated on the KDM model.

E. Data Collection and Interpretation

We verify, based on a set of oracle, whether all refactoring were successfully propagated throughout 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. In such tables there are two abbreviations: (i) "P.C?" ("Propagation Corrected?") and (ii) "N.A" ("Not Applied"). 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 KDM levels. Which means that in this case our approach could automatically execute truly relevant propagation in all KDM levels when dealing with the selected refactorings: *Extract Class*, *Move Class*, *Extract Layer*, and *Remove Class*. Observing globally the data in Table I we can argue that the use of our approach in the context of these refactorings has been satisfactory. The data also show that our approach it is able to propagate changes in KDM levels in an effective way and it also yields concise propagation of changes. Thereby, the **RQ**$_1$ can be answered as true, that is, the proposed approach can propagate changes effectively throughout all KDM levels.

VI. Evaluation

This section describes the experiment used to evaluate the effectiveness of our DI 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 DI 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: (*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 DI Algorithm, i.e., step [B] of our whole approach; (*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 DI Algorithm; (*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 summarized using Wohlin et al.'s template [15] as follows:

Analyze the effectiveness of our DI 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 DI Algorithm, step [B], in one system and compared the results with oracles and a 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.

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

- 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.

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

C. Experiment Desing

For our evaluation, we also used the same system described in Section V, LabSys. As stated before, firstly we transformed it into a KDM instance to apply our approach by means of MoDisco. Furthermore, to evaluate our DI Algorithm it was necessary to choose some refactoring. As a matter of fact, it is important to know its parameters once our DI Algorithm uses them to identify all affected meta-classes. 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 the DI Algorithm we counted whether all the affected meta-classes were successful identified.We also measured both software metrics precision and recall after applying the DI 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

Observing both Table II and Figure 6, it is possible to see that for the refactoring *Extract Class*'s parameters we got

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

Refactoring	**Extract Class**	**P.C?**	**Refactoring**	**Extract Layer**	**P.C?**
Code	Create an instance of ClassUnit that represent the new Class	Yes	Code	Create an instance of Package	Yes
	Move all StorableUnits to the new ClassUnit	Yes		Move a the selected ClassUnit from a Package to the new Package	Yes
	Move all MethodUnit to the new ClassUnit	Yes	Structure	Create an instance of Layer	Yes
	Create an intance of HasType, which represent an association between the new ClassUnit and the old ClassUnit	Yes		Create an instance of AggregationRelationship between the new Layer and the old one	Yes
Structure	N. A	N.A		Associate the new Layer by means of the association implementation with the new Package	Yes
Data	Create a instance of RelationalTable owning the name of the new ClassUnit	Yes		Summing up all primitive relationship to compute the meta-attribute density	Yes
	For each StorableUnit it is necessary to create a ItemUnit, which represent the RelationaTable columns.	Yes	Data	N. A	N. A
	Create an instance of UniqueKey that represent the primary key of the RelationalTable.	Yes	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
Conceptual	N. A	N. A			
Refactoring	**Move Class**	**P.C?**	**Refactoring**	**Remove Class**	**P.C?**
Code	Move an specific ClassUnti from a source Package to a target Package	Yes	Code	Delete the selected instance of a ClassUnit	Yes
Structure	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	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	N. A	N. A	Data	if the removed ClassUnit was associated with an instance of RelationalTable, then it should also be removed	Yes
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	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%

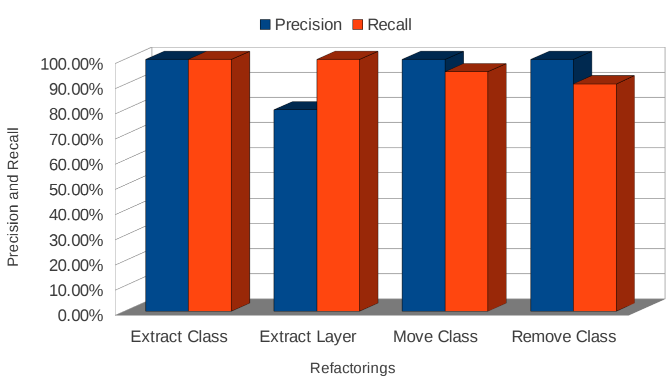

Fig. 6: Bar-plot for precision and recall of each refactoring.

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.

VII. Related Work

Westfechtel *et al*. [16] presented an approach for refactoring static models (UML class diagrams) and propagate the changes to behavioral models (UML Sequence diagrams), aiming to maintain the consistency between these models. Unlikely these authors, in this project our goal is to propagate the changes to other static views, all of them belonging to the same family of metamodels. Considering that our approach does take into consideration behavior aspects, only static ones, we believe that both approaches are complementary to each other.

Egyed [17] proposed an UML-based approach similar to our first step, which is the mining and identification of model elements to be changed. In order to find those model elements, the author employs ”consistency rules” between models. These rules always must keep satisfied when the models as synchronised/aligned. So, whenever an element is refactored, a broken rule is an indication that a desynchronisation problem occurred, allowing the identification of model elements that must be updated to synchronise the model again. The author argue that his approach scales up to large, industrial UML models. The author employs a strategy different from ours for the identification of the points to be updated; while we rely on the comparison between the original and refactored models, he relies on the consistency rules. We believe that the problem with his approach is the insertion of another task to be performed (the specification of the consistency rules), in which new problems and errors can be inserted. Our approach is more time-consuming in terms of processing, but we believe that the recall and precision of the identification is higher.

Therefore, to be best of our knowledge our work is the first one in presenting an approach for propagating changes in KDM models in a consistent and transparent way. The most fundamental differences of other related works are: i) we consider only static models, i.e, other views of the system; ii) we work with a family of metamodels that share a consistent and homogeneously terminology (syntax) and iii) our solution is KDM-specific and iv) our approach is tool supported by means of an Eclipse Plug-in which can be coupled to any refactoring scripts written in any transformation language.

VIII. Discussion

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, where all of them are somehow connected by means of associations. 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. However, as stated before, to the best of our knowledge, up to this moment, there is no research concentrated on investigating change propagation in KDM. We claim that by using our approach the modernization engineers can concentrate just on the development of the refactorings, without worrying about the change propagation, which is a time-consuming and error-prone task.

During the elaboration of this research we realized 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 [16]. 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.

IX. Conclusions and Future Work

The main contributions are: i) a Java mining algorithm to identify all KDM model elements that need to be updated when a specific refactoring is performed, ii) a propagation technique approach, and (iii) a support and preliminary infrastructure for allowing the creation of refactorings for kdm without worrying about he propagation of changes.

A possible future work is to integrate the proposed of Westfechtel *et al.* [16] with our presented approach.

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] 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.

[4] 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.

[5] S. Roser and B. Bauer, "Automatic generation and evolution of model transformations using ontology engineering space," in *Journal Data Semantics XI*. Springer Berlin Heidelberg, 2008, pp. 32–64.

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

[7] 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.

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

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

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

[11] H. K. Dam and M. Winikoff, "Supporting change propagation in uml models," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept 2010, pp. 1–10.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] A. Egyed, "Instant consistency checking for the uml," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 381–390.