Change Propagation-Aware KDM Refactorings

Rafael Serapilha Durelli[†], Fernando Chagas[*], 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,valter}@dc.ufscar.br
[†]Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: rdurelli@icmc.usp.br

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

I. Introduction

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

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

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

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

II. Motivation and running example

This section introduces an illustrative and motivating scenario that will be used as running example along each phase of the approach presented in this paper, and that will be detailed in Section IV. As a demonstration of the problem of propagation of changes in KDM levels/packages, let us consider a toy system that is based on a well know Model View Controller (MVC) design pattern. As noted in Figure 1 this system is split in four KDM levels/packages, which are illustrated in the figure bounded by dashed lines shape. Following is described each KDM levels/packages and its meaning regarding to the illustrated system.

- Code Package - represents the source-code (physical artefacts). In Figure 1 is is possible to see three packages: (*i*) GUI, (*ii*) CTR, and (*Model*). The first one, GUI, contains four classes, StudentGUI, InstructorGUI, Student, and Instructor. The second package contains three classes: StudentCTR, UtilCon, InstructorCTR. Then, the third one owns two classes, Secretary and Researcher. Note that these classes are related to each other by means of primitive relationships, such as: Calls, Creates, Extends, etc;
- Structure Package - illustrates the system's architecture, herein the system is based on MVC. As noted in Figure 1 each gray rectangle depicts a layer, i.e., View, Controller, and Model. For the first Layer, we have associated View with the package GUI. In KDM this kind of association is done by means of the meta-attribute implementation, which are depicted by the dashed arrows. Similarly, Controller was associated with the package CTR, and the Layer Model was associated with the package Model, respectively. Regarding to the relationships among Layers, let R (see Figure 1) be the corresponding aggregated relationship, which represents the number summing all primitive relationships among layers. For instance, the aggregation relationship between the layer "View" and

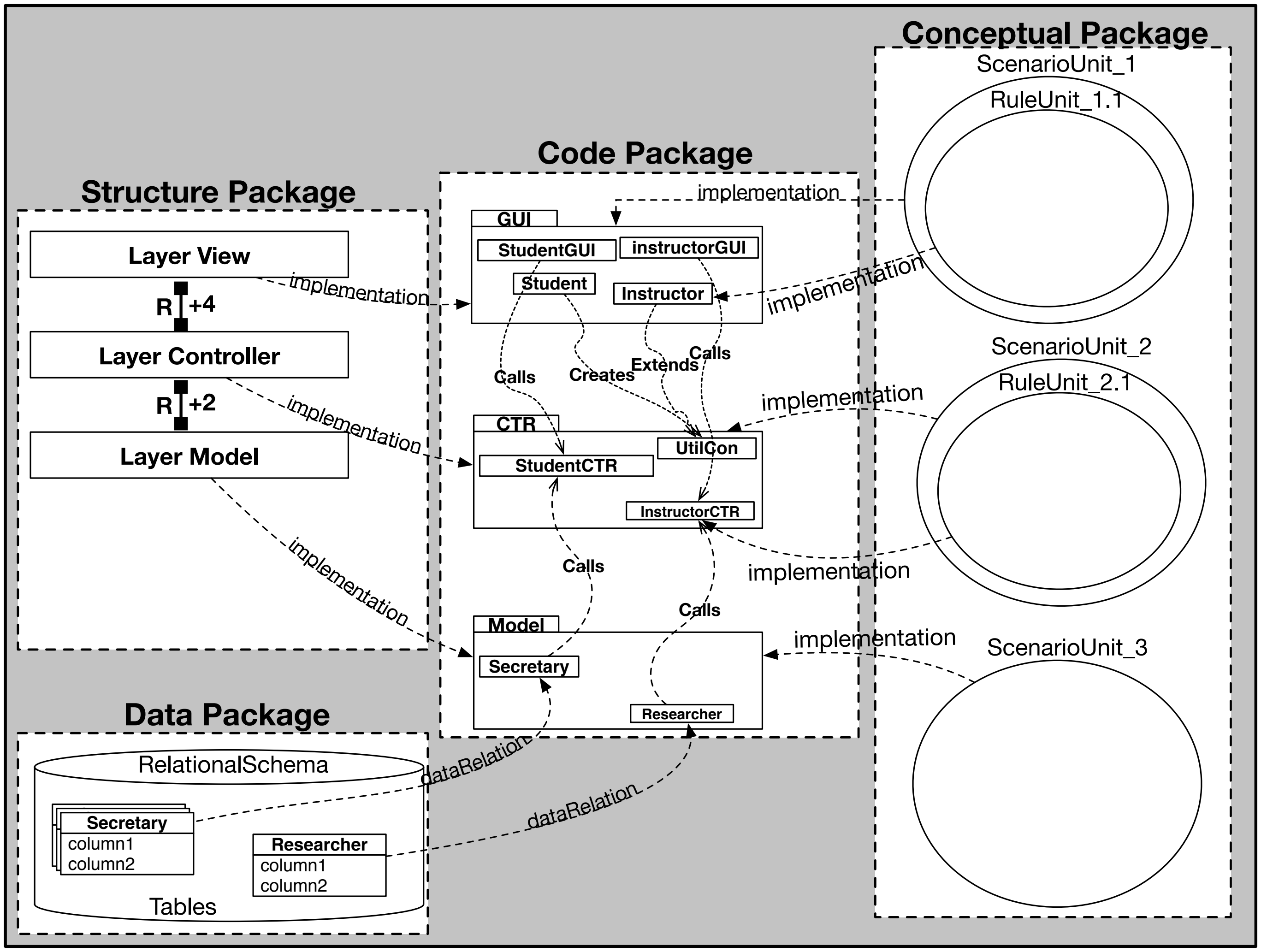


Fig. 1: Motivation and running example.

the layer "Controller" are represented by the relationships: "Calls", "Creates", "Extends", etc from the Code Package. Summing up these relationships R's value is 4. Following the same idea the relationship between the Layer Controller and Layer Model is 2;

- Conceptual Package - illustrates the system's business rules domain. Note that this system owns three scenarios, each of them are associated with a package from Code Package by means of the association implementation, see the dashed arrows. Further, each scenario contains a rule apart from the last one. In it turn, each rule is associated with a class from Code package, again using the association implementation;
- Data Package - depicts the system's database and its tables. Herein, it is possible to notice that the depicted system owns a set of Plain Old Java Objects (POJOS), they are: Student, Instructor, Secretary, and Researcher. All of these POJOS are also Object Relational Mapping (ORM), i.e., they are mapped to the Data package using the metaclasse RelationalTable.

Considering this system it is possible to highlight some problem or even to add new requirements. For instance, a problem that can be noticed is that both classes Student and Instructor should be contained in "model" package not in "GUI" package, respectively. In order to fix this problem, one should apply a refactoring - for instance, *Move Class*.

Concerning to a new requirement let's pretend someone has identified that the class Student is doing work that should be done by two classes - it contains attributes to hold informations upon student's addresses. In order to fulfill this new requirement one should apply the refactoring *Extract Class* and creates a new class named Address (which is a POJO and also an ORM) and move all Student's attributes related to address to this new class.

However, in both described refactoring it is necessary a skilled domain expert into KDM to identify all the metaclasses in the system which involve/reference the classes aforementioned and correct them respectively in all KDM packages, i.e., propagate all refactoring's impact throughout all KDM's packages.

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

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

Apparently, in a complex system comprising all kdm's packages/levels, propagate all changes after a refactoring is a difficult and error-prone task. Even identifying the affected parts of the KDM's packages/levels is not an easy and straightforward process. In order to fulfill and automatize this process we have devised an approach (Propagation-Aware Refactorings) that contains three main steps. The first step is the identification of all dependent elements related to a specific refactoring. In the second step the refactoring of all identified elements are performed using a model-to-model transformation language - the third step is the propagation of changes in order to keep all the dependent models synchronized. In the following sections we show the theoretical background need to fully understand our approach. Then, our approach is presented in Section IV.

III. Background

In this section we provide a brief background to Architecture-Driven Modernization (ADM) and Knowledge Discovery Metamodel (KDM). Further, we describe in detail why change propagation in KDM is a complex process.

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

B. Change Propagation in KDM

In our previous work [1], we introduced a refactoring catalogue for KDM for managing evolution of a software system. This paper served as a starting point to investigate how the changes affect the KDM's levels. For instance, depending on the refactoring a set of metaclasses must to be create, updated and even removed, these operations may cause minor or major changes to be propagated into other KDM's metaclasses. In order to explain the propagation of changes in KDM consider

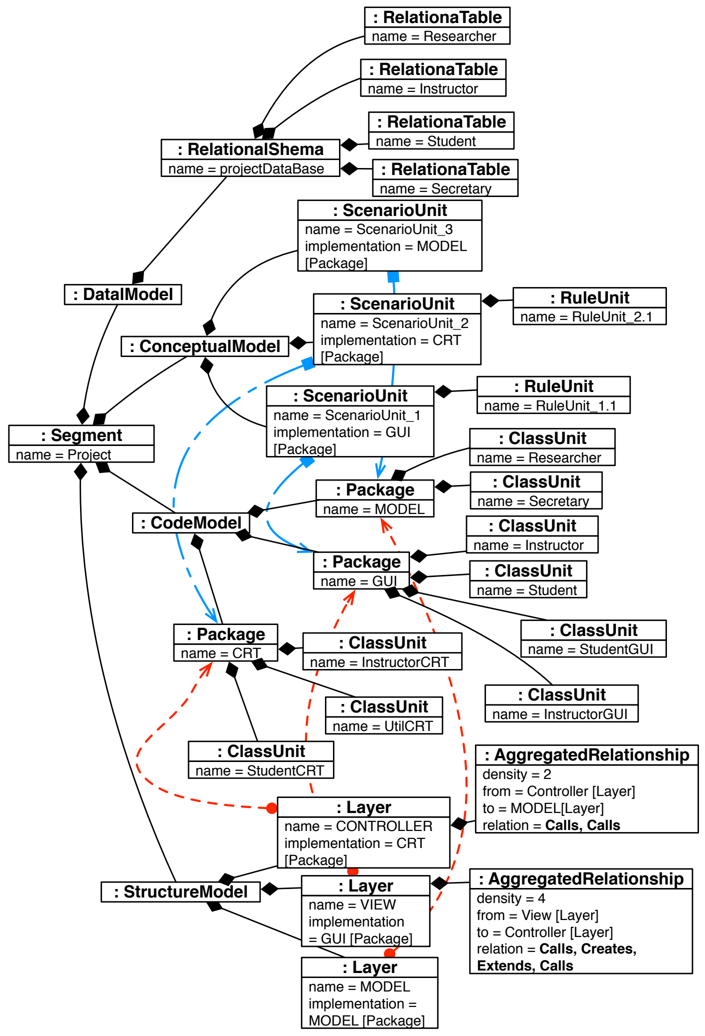


Fig. 2: A bird's eye view of a KDM's instance.

the Figure 2. This figure depicts the corresponding, though simplified KDM instance of system depicts in Figure 1. It illustrates a KDM instance as a UML object diagram for the sake of simplicity and due space limitations some elements are not depicted.

As we can see, a KDM's instance can be understood as a tree where we have a specially node called the root of the tree. Then the remaining nodes are partitioned into $M >= 0$ joint sets $T_1, T_2, ..., T_n$, and each of these sets is a subtree. Each nodes represent a metaclass that make up the system depicted in Figure 1. The edges represent the relationship between the metaclasses.

The root is the metaclass `Segment`. There are four subtrees rooted at `StructureModel`, `CodeModel`, `ConceptualModel`, and `DataModel`, respectively. The tree rooted at `StructureModel` has three `Layers`, `CONTROLLER`, `VIEW`, and `MODEL` - they are connected by the metaclasses `AggregatedRelationship` (see Figure 1 and Figure 2).

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

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

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

As we can see in Figure2, there are not only horizontally relations between the models, but the elements of the system can also be vertical related across the vertical partitions. A few examples are denoted by the red/blue dashed arrows. For instance, there is a relation between a `CodeModel` (its respective metaclasses) with the `StructureModel` - which means that a change in one of the ends of the relation can influences the other.

Considering these KDM's models leads to evolution of each affected model separately. However, this is a highly time-consuming and error-prone solution since we need a domain expert who is able to identify all the affected models and propagate the changes. Following we present our approach to detect and propagate all the changes throughout all KDM's levels.

IV. Propagation-Aware Refactorings

In order to fulfill the limitation pointed out in Section III-B, we introduce an approach that aims to propagate all the changes throughout the KDM's levels. It ensures that whenever a change/refactoring is performed in any KDM's level, it is correctly propagated to affected KDM's levels and vice versa. So it ensures consistency between the KDM's levels when they are refactored.

More specifically, our approach is divided in three steps, which are depicted by its corresponding letters and tittle at the left side in Figure **??**.

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

In step [B], Apply Refactoring, the software engineer should perform a set of refactorings. In this step, new metaclasses can be created, updated, and removed. In this step we have used model-to-model (M2M) transformation language to perform the refactorings.

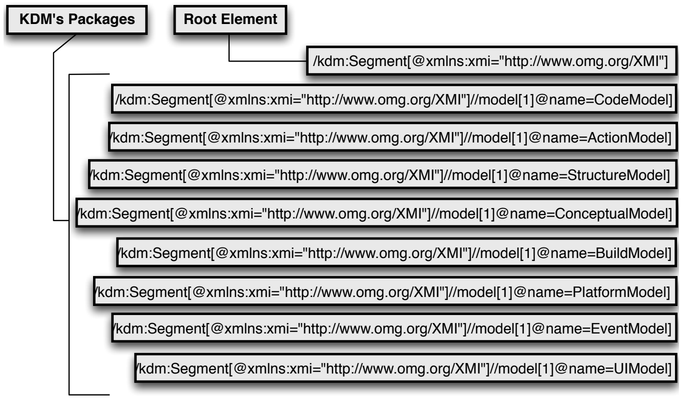
In step [C], Update KDM Instance, involves updating the elements identified in the step [A]. As in step [B], in this step we also have used M2M to update all KDM's instances. More details on each step are provided in the next sections.

A. Mine Affected Metaclasses

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

Let us consider the running example depicted in Figure 1. In this example, the engineer aims to apply the refactoring *Move Class* - both classes `Student` and `Instructor` should not longer be contained in the package `View`. These classes should be allocated into the package `Model`. Considering the refactoring *Move Class*, three elements (`Student`, `Instructor`, and their package) need to be investigated throughout the KDM's instance in order to identify propagation scenarios of changes.

Therefore, firstly a query must be executed to get the root elements in KDM. This query is represented as the first statement in Figure 5 - 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 5. The returned elements of these queries are used as input in our depth-first search algorithm.



```
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=CodeModel]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=ActionModel]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=StructureModel]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=ConceptualModel]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=BuildModel]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=PlatformModel]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=EventModel]
/kdm:Segment[@xmlns:xmi="http://www.omg.org/XMI"]//model[1]@name=UIModel]
```

Fig. 3: Xpath used to return the KDM's root element, Segment.

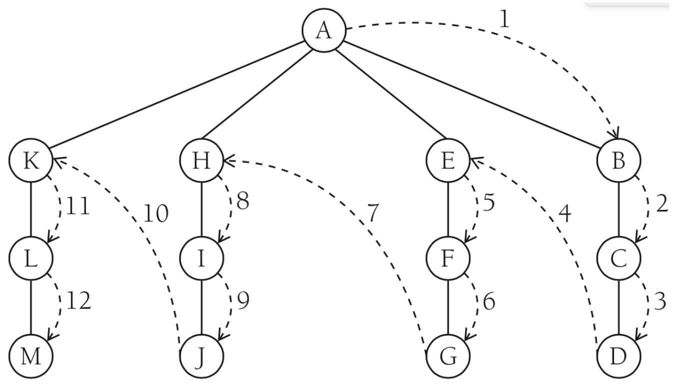


Fig. 4: Depth-First Search.

Algorithm 1 depicts the depth-first search algorithm that is used to mine all the affected metaclasses. It takes as input a Segment, the elements that will be refactored in Step [B] (e.g., for the refactoring `Move Class` three affected elements - `Student`, `Instructor`, and their package), and all KDM's levels recuperated by the queries depicted in Figure 5.

A diagram of how our depth-first search algorithm works is shown in Figure 4. Each node represents a metaclass and the edges represent the relationship among the metaclasses. More specifically, the algorithm works as: 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 you reach the last metaclass. Then you check to see if the `Segment` has any unvisited adjacent metaclass. If it does not, then you pop it off the stack and check the next metaclass. If you find one, you start 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 you 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.

As can be visualised, a stack is used to store all affected elements, see Algorithm 1 line 2, ❶. In line 3 a generic KDM element is defined. While *seg* is non-empty, a node is chosen for expansion (line 4).

As every element, except the Segment, is connected somehow it is necessary to iterate throughout them, line 4 of Algorithm 2 illustrates this iteration. After, the method `isAffected(...)` is called to verify if the element is affected. If the condition in line 8 evaluates to true, then the element is pushed into the stack defined in line 2. Finally, in line 20 the stack is returned with all affected elements, see Algorithm 2 ❸.

Algorithm 1: Depth-First Search Algorithm.

```
   Input: KDMEntity kdmElement, Segment segment,
          KDMModel model
   Output: All affected metaclasses
1  begin
2      $Stack stack \longleftarrow \{\}$;
3      KDMEntity elementToVerify;
4      foreach seg in segment do
5          if seg.getOwnedElements() != null then
6              if seg.getNextSiblind() != null then
7                  $elementToVerify \longleftarrow$
                   $seg.getNextSiblind()$;
8                  if ❶ isAffected(elementToVerify,
                   kdmElement, model) then
9                      stack.push(elementToVerify);
10                     $seg \longleftarrow seg.getFirstChild()$;
11                 end
12             end
13         else
14             $seg \longleftarrow seg.getNextSiblind()$;
15             if seg = null && stack.isEmpty() then
16                 // return to the parent's level
17             end
18         end
19     end
20     ❸ return stack
21 end
```

B. Apply Refactoring

In the step [B] the engineer must apply the refactoring. A natural way of implementing refactoring in models is by means of *in-place transformations*[1]

In-place transformations can be described in many ways. Rule-based descriptions are elegant and easy to understand. Such descriptions have declarative model rewriting rules as their primitive building blocks. A rule consist of a *Left Hand Side* (LHS) pattern that is matched against a model. If a match is found, this pattern is updated in the model, based on what is specified in the *Right Hand Side* (RHS) of the rule.

We have devised a repository where a set of *in-place transformations* (i.e., refactoring) is available[2]. All the *in-place transformations* can be written either in ATL Transformation Language (ATL) or Query/View/Transformation (QVT).

Considering again the running example presented in Section II, where the Extract Package refactoring must be applied. the engineer must browser our repository and choose the refactoring Extract Package.

Refactorings are implemented by specifying its actions that have to be executed for applying the change. Most of them

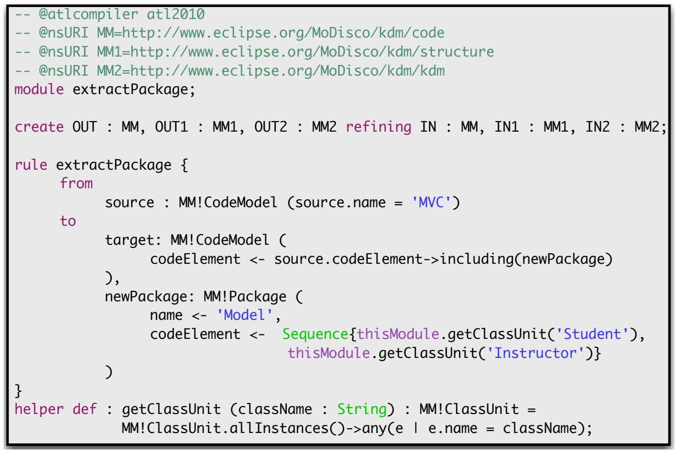
[1]The term *in-place transformations* stands for transformations rewriting a model, as opposed to producing a model from scratch which is done by *out-place transformations*.

[2]The repository can be accessed in www.site.com.br. It aims is to share refactoring to be applied into KDM's instances

Algorithm 2: isAffected Algorithm

```
Input: KDMEntity kdmElement, KDMModel model,
       KDMEntity e
Output: true or false
begin
    if (e = AbstractUIElement) or (e = AbstractStructureElement) or (e = BuildResource) or
    (e = AbstractPlatformElement) or (e = AbstractConceptualElement) or (e = AbstractEventElement) then
        foreach elements in e.getImplementation() do
            if elements = ele then
                return true
            end
        end
    end
    else if e = AbstractDataElement then
        foreach elements in elementToV.getDataRelation() do
            if elements = elementToVerify then
                return true
            end
        end
    ...
end
```

need also some input parameters that should be properly instantiated by the user.



```
-- @atlcompiler atl2010
-- @nsURI MM=http://www.eclipse.org/MoDisco/kdm/code
-- @nsURI MM1=http://www.eclipse.org/MoDisco/kdm/structure
-- @nsURI MM2=http://www.eclipse.org/MoDisco/kdm/kdm
module extractPackage;

create OUT : MM, OUT1 : MM1, OUT2 : MM2 refining IN : MM, IN1 : MM1, IN2 : MM2;

rule extractPackage {
    from
        source : MM!CodeModel (source.name = 'MVC')
    to
        target: MM!CodeModel (
            codeElement <- source.codeElement->including(newPackage)
        ),
        newPackage: MM!Package (
            name <- 'Model',
            codeElement <-  Sequence{thisModule.getClassUnit('Student'),
                             thisModule.getClassUnit('Instructor')}
        )
}
helper def : getClassUnit (className : String) : MM!ClassUnit =
          MM!ClassUnit.allInstances()->any(e | e.name = className);
```

Fig. 5: Chunk of code in ATL to perform the refactoring Extract Package.

V. Related Work

The approach proposed by Cechticky *et al.* [2] allows object-oriented application framework reuse by using a tool called OBS Instantiation Environment. That tool supports graphical models do define the settings of the expected application to be generated. The model to code transformation generates a new application that reuses the framework.

The proposal found in this paper differs from their approach on the following topics: 1) their approach is restricted to frameworks known during the development of the tool; 2) it does not use aspect-orientation; 3) the reuse process is applied on application frameworks, which are used to create new applications.

Another approach was proposed by Oliveira *et al.* [3]. Their approach can be applied to a greater number of object oriented frameworks. After the framework development, the framework developer may use the approach to ease the reuse by writing the cookbook in a formal language known as Reuse Definition Language (RDL) which also can be used to generate the source code. This process allows to select the variabilities and resources during reuse, as long as the framework engineer specifies the RDL code correctly.

These approaches were created to support the reuse during the final development stages. Therefore, the approach proposed in this paper differs from others by the supporting earlier development phases. This allows the application engineer to initiate the reuse process since the analysis phase while developing an application compatible to the reused frameworks. Although the approach proposed by Cechticky *et al.* [2] is specific for only one framework, its can be employed since the design phase. The other related approach can be employed in a higher number of frameworks, however it is used in a lower abstraction level, and does not support the design phase. Other difference is the generation of aspect-oriented code, which improves code modularization.

VI. Conclusions

Acknowledgments

The authors would like to thank CNPq for funding (Processes 132996/2010-3 and 560241/2010-0) and for the Universal Project (Process Number 483106/2009-7) in which this paper was created. Thiago Gottardi would also like to thank FAPESP (Process 2011/04064-8).

References

[1] 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.

[2] V. Cechticky, P. Chevalley, A. Pasetti, and W. Schaufelberger, “A generative approach to framework instantiation,” in *Proceedings of the 2nd international conference on Generative programming and component engineering*, ser. GPCE ’03. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 267–286. [Online]. Available: http://portal.acm.org/citation.cfm?id=954186.954203

[3] T. C. Oliveira, P. Alencar, and D. Cowan, “Reusetool-an extensible tool support for object-oriented framework reuse,” *J. Syst. Softw.*, vol. 84, no. 12, pp. 2234–2252, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2011.06.030