

Change Propagation-Aware KDM Refactorings

Rafael Serapilha Durelli[†], Fernando Chagas*, Bruno M. Santos*,
Marcio Eduardo Delamaro[†] and Valter Vieira de Camargo*

*Departamento de Computação, Universidade Federal de São Carlos,
Caixa Postal 676 – 13.565-905, São Carlos – SP – Brazil

Email: {fernando_chagas, bruno.santos, valter}@dc.ufscar.br

[†]Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: rdurelli@icmc.usp.br

Abstract—Architecture-Driven Modernization (ADM) is a model-driven alternative to conventional reengineering processes that relies on the Knowledge-Discovery Metamodel (KDM) as the base for the whole process. Unlike conventional metamodels, KDM is capable of putting together different system abstractions (Code, Architecture, Conceptual models) in a 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 synchronised [, , ,]

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

II. BACKGROUND

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

A. ADM and KDM

The growing interest in using Model-Driven Development (MDD) to manage software evolution is mainly focused on the reengineering or modernization of legacy systems. Several software migration projects have been carried out with model-driven approaches [?], [?], [?].

This growing interest motivated OMG to define the ADM initiative [?] which advocates carrying out the reengineering process considering MDD principles. ADM is the concept of modernizing existing systems with a focus on all aspects of the current systems architecture and the ability to transform current architectures to target architectures by using all principles of MDD [?, p. 60].

Figure 1 depicts the horseshoe model (i.e., horseshoe is basically a left-hand side, a right-hand side and a bridge between the sides) which was adapted to ADM.

- **Reverse Engineering:** herein a reverse engineering is realized, it takes a legacy system to be modernized as input, then the knowledge is extracted and a PSM is generated. In addition the PSM serves as the basis for the generation of a Platform-Independent Language (PIM), which is called KDM;
- **Restructuring:** in this phase a set of restructuring/refactoring can be applied into a KDM's instance by means of M2M transformations;
- **Forward Engineering:** then a forward engineering is carried out and the source code of the modernized target system is generated.

In order to perform a systematic modernization as depicted in Figure 1, ADM introduces several modernization standards,

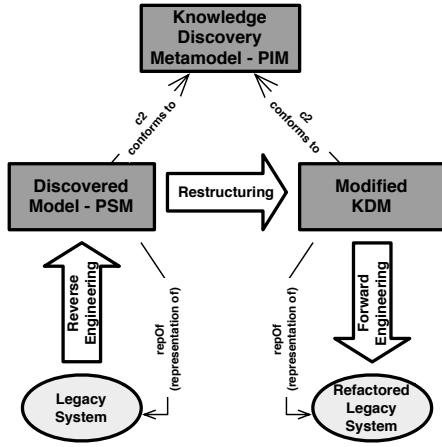


Fig. 1: Horseshoe Modernization Model. This figure is adapted from [?].

among them there is the Knowledge Discovery Metamodel (KDM). KDM is an OMG specification adopted as ISO/IEC 19506 by the International Standards Organization for representing information related to existing software systems. The goal of the KDM standard is to define a metamodel to represent all the different legacy software artifacts involved in a legacy information system (e.g. source code, user interfaces, databases, business rules, etc.).

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

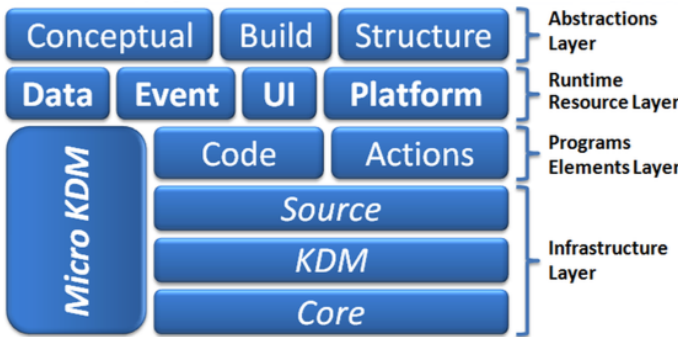


Fig. 2: KDM Architecture.

Although KDM is a metamodel to represent a whole system, its main purpose is not the representation of models related

strictly to the source code nature such as Unified Modeling Language (UML). While UML can be used to generate new code in a top-down manner, an ADM-based process using KDM starts from the different legacy software artifacts and builds higher-abstraction level models in a bottom-up manner through reverse engineering techniques.

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

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

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

Next, the Structure Package illustrates the system's architecture, herein the system is based on MVC. As noted in Figure 3 each rectangle depicts a layer, i.e., View ❹, Controller ❺, and Model ❻. The View layer is realized in source-code level by the package GUI; the Model layer is realized by the package Model and the layer Controller by the CTR package. These realizations are represented in KDM by the implementation relationship, represented in the figure by dashed arrows. Regarding to the relationships among the layers, it is possible to visualize pipes between two layers (see Figure 3. These pipes represents the corresponding AggregatedRelationship¹, which represents the number summing all primitive relationships among layers. For instance, the AggregatedRelationship between the layer View and the layer Controller are represented by the relationships: Calls, Creates, Extends, and another Calls from the Code Package. Summing up these relationships the density value is 4. Following the same idea the relationship between the layer Controller and layer Model is 2;

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

Finally, the Data Package depicts the system's database and its tables. Herein, it is possible to notice that the depicted system owns a set of Plain Old Java Objects (POJOs), they

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

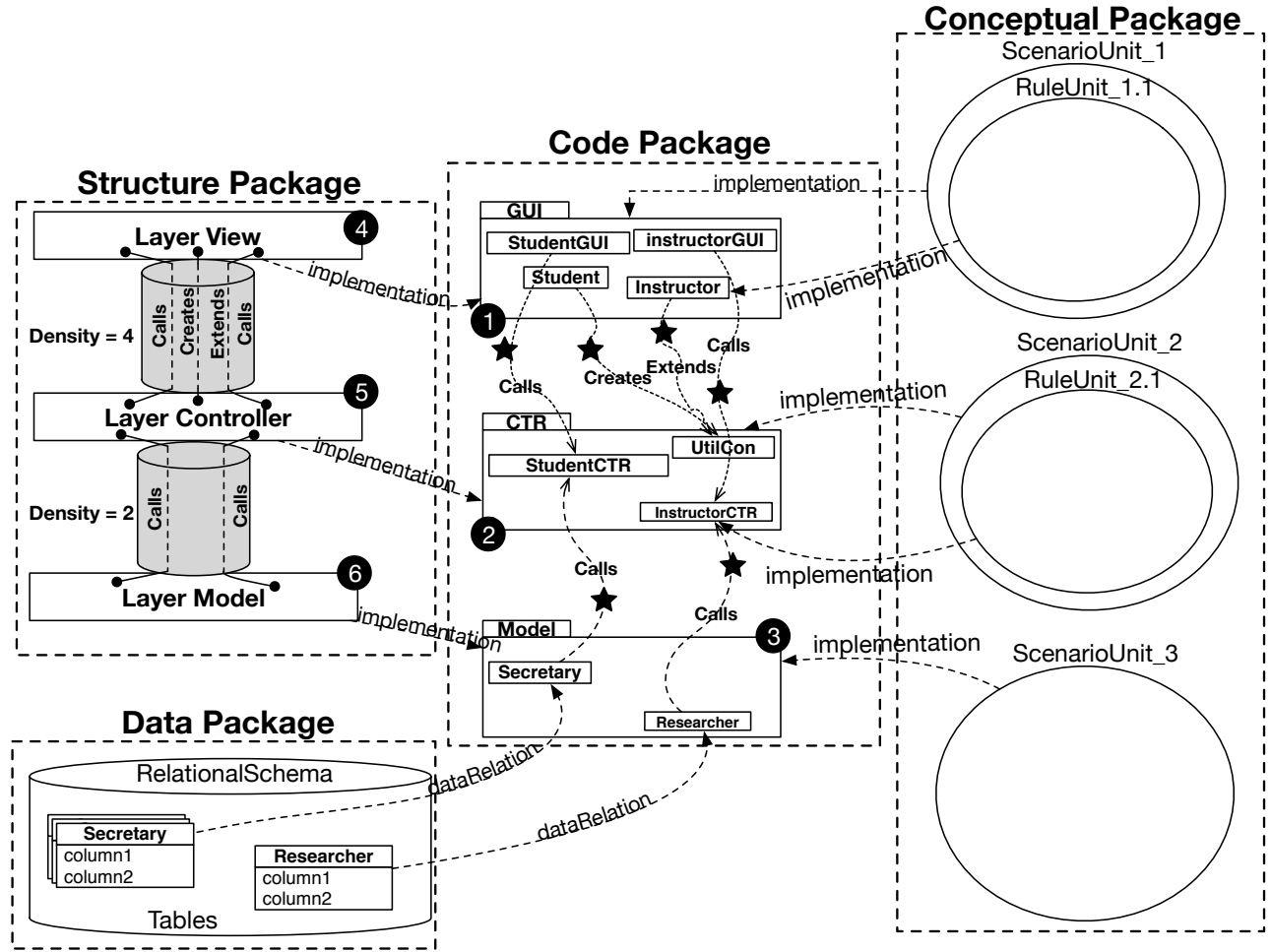


Fig. 3: Motivation and running example.

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.

The system described can be realized in KDM. For instance, the Figure 4 depicts the corresponding, though simplified KDM instance of system depicts in Figure 3. It illustrates a KDM instance as a UML object diagram for the sake of simplicity. Notice that due space limitations some elements are not depicted in this figure.

A KDM's instance can be understood as a tree where we have a specially node called the root of the tree. Then the remaining nodes are partitioned into $M \geq 0$ joint sets T_1, T_2, \dots, T_n , and each of these sets is a subtree. Each nodes represent a metaclass that make up the system depicted in Figure 3. The edges represent the relationship between the metaclasses.

The root is the metaclass Segment. There are four subtrees rooted at StructureModel, CodeModel, ConceptualModel, and DataModel, respectively. The tree rooted at StructureModel has three Layers, CONTROLLER, VIEW, and MODEL - they are connected

by the metaclasses AggregatedRelationship (see Figure 3 and Figure 4).

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

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

III. CHANGE PROPAGATION IN KDM

In our previous work [?], 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.

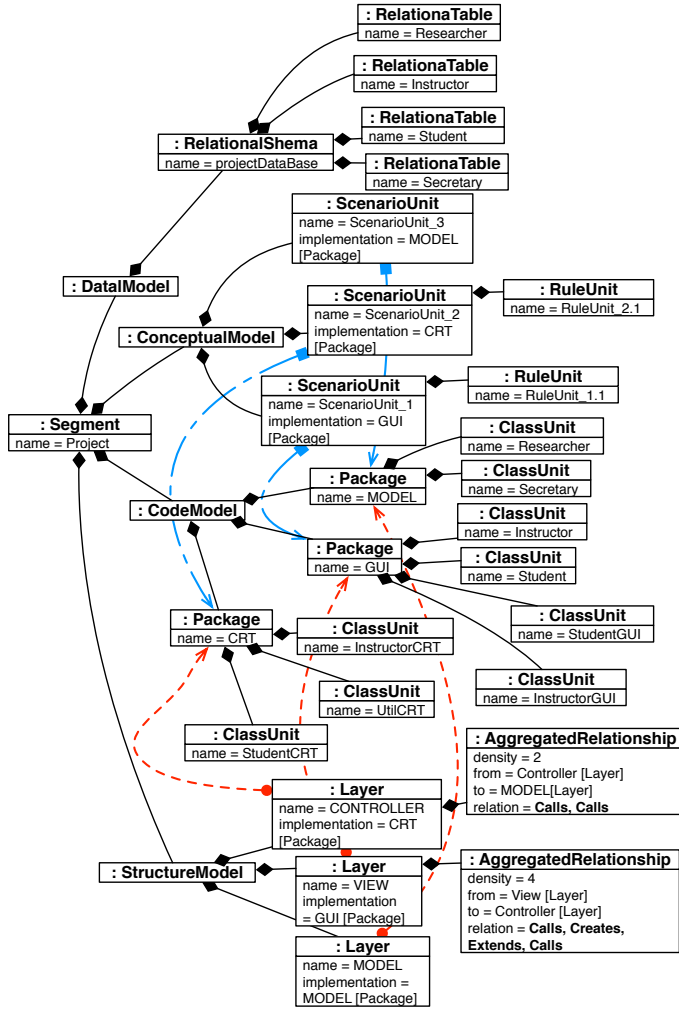


Fig. 4: A bird's eye view of a KDM's instance.

Considering the system described earlier it is possible to highlight some problem or even to add new requirements that will propagate changes at KDM's levels. For instance, a problem that can be noticed is that both classes *Student* and *Instructor* should be contained in *Model* package not in *GUI* package, respectively. In order to fix this problem, one should apply the refactoring *Move Class*. Regarding to a new requirement let's pretend someone has identified that the class *Student* is doing work that should be done by two classes - 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.

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 relationship 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*.

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

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

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

Considering these KDM's models 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.

In order to fulfill this limitation and create an automa-

tized process we have devised Propagation-Aware Refactorings (PAREf) that contains three main steps. The first step is the identification of all dependent elements related to a specific refactoring. In the second step the refactoring of all identified elements are performed using a model-to-model transformation language - the third step is the propagation of changes in order to keep all the dependent models synchronized. In the following sections our approach is presented.

IV. PROPAGATION-AWARE REFACTORINGS

In order to fulfill the limitation pointed out in Section ??, 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 title in Figure 5.

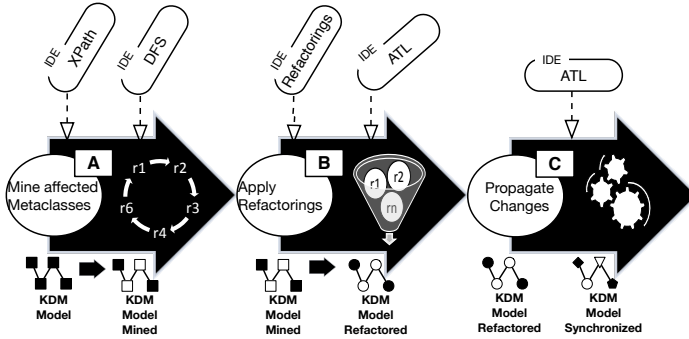


Fig. 5: Propagation-Aware Refactorings steps.

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

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

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

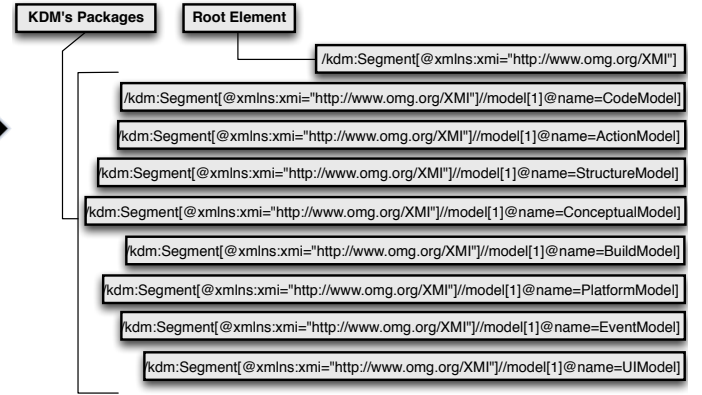


Fig. 6: Xpath used to return the KDM's root element, Segment.

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

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

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
5         then
6           foreach implementations element do
7             verify all elements in implementation
8           end
9           Mark vertex v as visited (via edge e).
10          Recursively call DFS (G, v).
11        end
12      end
13    end
14  end

```

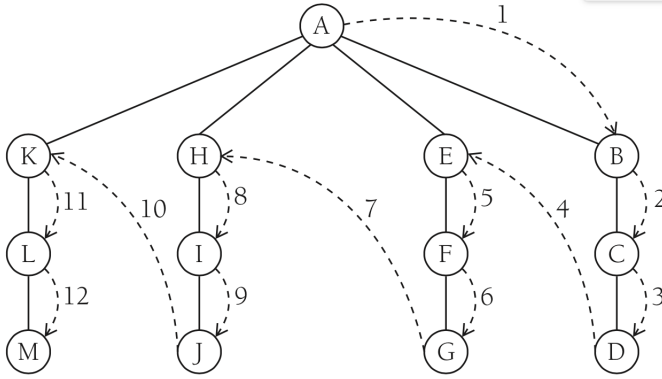


Fig. 7: Depth-First Search.

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. A diagram of how our depth-first search algorithm works is shown in Figure 7. Each node represents a metaclass and the edges represent the relationship among the metaclasses.

B. Apply Refactoring

In the step [B] the engineer must apply the refactoring. A natural way of implementing refactoring in models is by means of *in-place transformations*, which is illustrated in Figure 8. This kind of transformations are used for rewriting a model by creating, deleting, and updating elements in the input model, e.g., in a KDM's instance. 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 [?]:

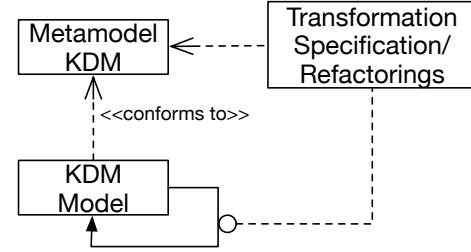


Fig. 8: Model transformation: in-place.

A natural way of implementing refactoring in models is by means of *in-place transformations*²

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 available³. All the *in-place transformations* can be written either in ATL Transformation Language (ATL) or Query/View/Transformation (QVT). Due space limitation this repository is not shown. However, the reader should keep in mind from where we get the in-place transformations. Considering again the running example presented in Section III, where the *Move Class* refactoring must be applied. Then the engineer must browser our repository and choose the refactoring *Move Class*.

```

-- @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. 9: Chunk of code in ATL to perform the refactoring Extract Package.

Refactorings are implemented by specifying its actions that have to be executed for applying the change. Most of them need also some input parameters that should be properly

²The 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*.

³The repository can be accessed in www.site.com.br. It aims is to share refactoring to be applied into KDM's instances

instantiated by the user. For instance, consider the chunk of code written in ATL depicted in Figure 9

V. PROOF-OF-CONCEPT IMPLEMENTATION

We devised a Eclipse plug-in named Modernization-Integrated Environment (MIE) which is split in three layers, as follows: (i) Core Framework, (ii) Tool Core, and (iii) Graphical User Interface (GUI). This plugin was devised on the top of the Eclipse Platform; The first layer we used both Java and Groovy as programming language. Moreover, the Core Framework layer contains a set of Eclipse plug-ins on which our environment is based on, such as MoDisco and Eclipse Modeling Framework (EMF)⁴. We used MoDisco⁵ once it is an extensible framework to develop model-driven tools to support use-cases of existing software modernization and provides an Application Programming Interface - (API) to easily access the KDM model. Also, EMF was used to load and navigate KDM models that were generated with MoDisco. The second layer, the Tool Core, is where the steps presented in Section IV were implemented. Herein, we work intensively with KDM models, which are XML files. Therefore, we use XPath to handle those types of files, to mine the affected metaclasses, ATL to perform the refactoring and to propagated them. Finally, the third layer is the Graphical User Interface (GUI) that consists of a set of SWT windows with several options to perform the refactorings based on the KDM model.

VI. RELATED WORK

In [?], 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.

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

VII. CONCLUSIONS

ACKNOWLEDGMENTS

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

REFERENCES

- [1] R. Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos, and L. Andrade, "Architectural transformations: From legacy to three-tier and services," Springer Berlin Heidelberg, 2008, pp. 139–170.
- [2] L. Andrade, J. a. Gouveia, M. Antunes, M. El-Ramly, and G. Koutsoukos, "Forms2net & #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] T. Reus, H. Geers, and A. van Deursen, "Harvesting software systems for mda-based reengineering," in *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 213–225.
- [4] R. Perez-Castillo, I. G.-R. de Guzman, O. Avila-Garcia, and M. Piattini, "On the use of adm to contextualize data on legacy source code for software modernization," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 128–132. [Online]. Available: <http://dx.doi.org/10.1109/WCRE.2009.20>
- [5] W. M. Ulrich and P. Newcomb, *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.
- [6] 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.
- [7] R. F. P. Boris Gruschko, Dimitrios S. Kolovos, "Towards synchronizing models with evolving metamodels," in *In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.
- [8] 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.
- [9] 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.

⁴<http://www.eclipse.org/modeling/emf/>

⁵<http://www.eclipse.org/MoDisco/>

- [10] K. Rui and G. Butler, "Refactoring use case models: The metamodel," in *Proceedings of the 26th Australasian Computer Science Conference - Volume 16*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 301–308.
- [11] J. Zhang, Y. Lin, and J. Gray, "Generic and domain-specific model refactoring using a model transformation engine," in *Volume II of Research and Practice in Software Engineering*. Springer, 2005, pp. 199–218.
- [12] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, 2000, pp. 154–164.