KDM-RE: A Model-Driven Refactoring Tool for KDM

Rafael S. Durelli[1], Daniel S. M. Santibáñez[2] , Márcio E. Delamaro[1] and Valter V. de Camargo[2]

[1]Computer Systems Department University of São Paulo - ICMC
São Carlos, SP, Brazil.

[2]Computing Departament
Federal University of São Carlos - UFSCAR
São Carlos, SP, Brazil.

{rdurelli, delamaro}@icmc.usp.br[1], valter@dc.ufscar.br[2]

Abstract. *Software refactoring is a proven technique that aims at improving the quality of source code. However, nowadays with the advent of Architecture-Driven Modernization (ADM), which follows the Model-Driven Architecture (MDA) principles and uses the Knowledge Discovery Metamodel (KDM), it seems a good alternative for moving from software refactoring to model-driven refactoring in order to create a language and platform independent catalog of refactoring. Nevertheless, although ADM provides the process for refactoring legacy systems, is so far missing an Integrated Development Environments (IDEs) to lead engineers to automatically apply refactorings as such exist in others object-oriented languages. We describe a tool, implemented as an Eclipse plug-in designed to fulfill exactly this need. This tool supports 16 refactorings that are heavily inspired by refactorings well known in the literature and yet keeping them in synch with the underlying source code. Our tool involves three steps: (i) reverse engineering, (ii) refactoring by means of KDM, and (iii) forward engineering. The first step relies on transforming the source code into a KDM's instance. The second step relies on applying refactorings into the KDM's instance. The third step consists of generating the refactored source-code.*

1. Introduction

Architecture-Driven Modernization (ADM) has been proposed by Object Management Group (OMG) in order to carry out refactoring following the principles of Model-Driven Architecture (MDA) [Kleppe et al., 2003] has become a great candidate since it aims to promote *(i)* portability, *(ii)* interoperability, and *(iii)* reusability by means of model. According to the OMG the most important artifact provided by ADM 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. KDM is structured in a hierarchy of four layers; *Infrastructure Layer*, *Program Elements Layer*, *Runtime Resource Layer*, and *Abstractions Layer*. We are specially interested in the *Program Elements Layer* because it defines the Code and Action packages which are widely used by our tool. The Code package defines a set metaclasses that represents the common elements in the source code supported by different programming languages such as: (*i*) `ClassUnit` and `InterfaceUnit` which

represent classes and interface, respectively, (*ii*) `StorableUnit` which illustrates attributes and (*iii*) `MethodUnit` to represent methods, etc. The Action package represent behavior descriptions and control-and-data-flow relationships between code elements.

On the other hand, refactoring has been known and highly used both industrially and academically. It is a form of transformation that was initially defined by Opdyke [Opdyke, 1992] as "a change made to the internal structure of the software while preserving its external behavior at the same level of abstraction". Nowadays it is possible to identify several catalogs of refactoring for different languages and the most complete and influential was published by Fowler in [Fowler et al., 2000]. However, while software reengineers would greatly benefit from the possibility to assess different choices during the refactorings, in practice they mostly rely on experience or intuition because of the lack of approaches providing comparison between possible variations of a change. Therefore, software reengineers do not have the possibility to easily apply analyses on different source-code version branches of a system and compare them to pick up the most adequate changes. In this context, the motivations for moving from software refactoring to model-driven refactoring are: (*i*) a model provides an abstract view of the system, hence, visualizations of the structural changes required are easier, (*ii*) problems uncovered at the design-level can be improved directly on the model, and (*iii*) using refactoring in high abstract level can allow the software engineer to explore alternate design paths in much cheaper than software refactoring.

To overcome the described problems, we devised a *plug-in* on the top of the Eclipse Platform named **K**nowledge **D**iscovery **M**odel-**R**efactoring **E**nvironment (KDM-RE). This *plug-in* provides an environment for model-driven refactoring based on KDM models, i.e., we provide an MDA *plug-in* with the ability to improve existing KDM models, yet keeping them in synch with the underlying source code. In addition, this *plug-in* supplies a multiple versions of a system at level models (KDM), enabling to the engineer to work interactively on multiple models and to explore alternate refactoring path. Finally, after the best alternate refactoring path is chosen by the engineer, a forward engineering can be performed and the refactored source code is generated. Notice that KDM-RE supports 16 refactorings that are heavily inspired by the refactorings given by Fowler [Fowler et al., 2000]. This paper is organized as followed: Section 2 provides information related to KDM-RE - Section 3 the architecture of KDM-RE is depicted - in Section 4 there are related works and in Section 5 we conclude the paper with some remarks and future directions.

2. KDM-RE

Figure 1 depicts the overall process to use KDM-RE - which was adapted based on the horseshoe modernization model. It is split into three steps, they are: (*i*) reverse engineering, (*ii*) refactorings, and (*iii*) forward engineering. Furthermore, these steps were divided into six levels, as can be seen in Figure 1 - more details about these steps and its levels are described in next sections:

2.1. Reverse Engineering

Herein the engineer can provide a KDM file or a source-code from a legacy system. If the engineer provides a KDM file then KDM-RE can already apply the refactorings. In

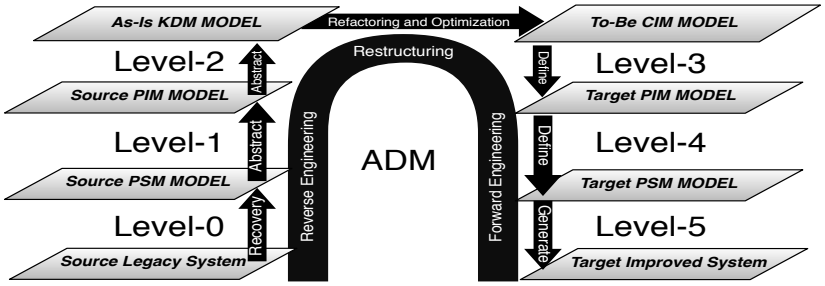

Figure 1. KDM-RE Process

case of the source code is the input, then the engineer starts the process in the **Level-0** by choosing an eclipse project which contains the source-code to realize the refactorings. After that, into **Level-1** the source-code need to be transformed into a Platform-Specific Model (PSM). This PSM is an instance of the source-code which represents all abstraction of the source-code. To realize this transformation we implemented a model extractor in Java. Figure 2 illustrates how KDM-RE manages to assist the engineer to get the instance of this PSM. Figure 2 (a) shows the eclipse project selected by the engineer - then after the engineer click in a popup menu named "Discovery Java Model" the Figure 2 (b) is shown an excerpt to indicate the correspondence with the legacy Java Model. For instance, each "Class" found in the source-code an instance of the meta-classe `ClassDeclaration` is created, similar each "Methods" declarations are transformed to instances of the meta-classe `MethodDeclaration`, and "attributes" are transformed into instances of `FieldDeclaration`, etc.

After creating the PSM the next level (**Level-2**) consists in transforming the PSM to a Platform-Indented Model (PIM) which is based on the KDM. In this level the KDM-RE uses MoDisco[1] which provides an extensible framework to transform a specific PSM to KDM models in order to represent the systems "AS-IS". We added in KDM-RE a popup menu named "Discovery KDM Model" which by clicking on it the KDM-RE calls the MoDisco API to get a instance of the KDM based on the earlier PSM. In Figure 2(c) shows how the Java Model is transformed to KDM. In Figure 3 we depicted the main window of our *plug-in*. For explanation purpose, we have identified main regions, i.e., ⓐ, ⓑ, ⓒ and ⓓ.

As already stated all refactorings provided by KDM-RE are based on the KDM model. In order to assist the refactorings we extended the KDM's model browser provided by MoDisco. We added a popup menu named "Refactoring KDM" in this model browser, see Figure 3ⓐ. By using this menu the engineer can interact with the KDM model and choose which refactoring must be carried out in the KDM. In the region ⓐ can be seen all 16 refactorings that have been implemented in KDM-RE. For illustration purposes only we drew rectangles to separate the refactorings into three groups. The black rectangle represents refactorings that deal with generalization, the blue rectangle stand for refactorings to organize data and the red one symbolize refactoring to assist the moving features between objects.

The region ⓑ on Figure 3 shows an UML class diagram that can be used either

[1] `http://www.eclipse.org/MoDisco/`

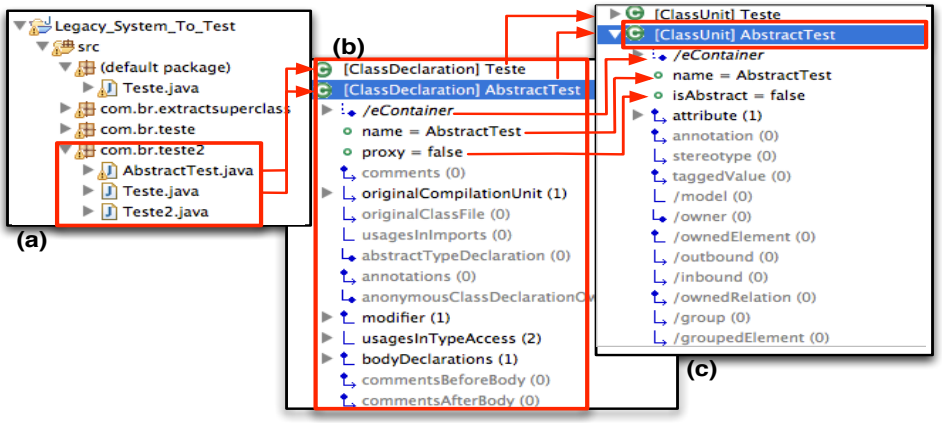

Figure 2. Process to Discovery Java and KDM Model

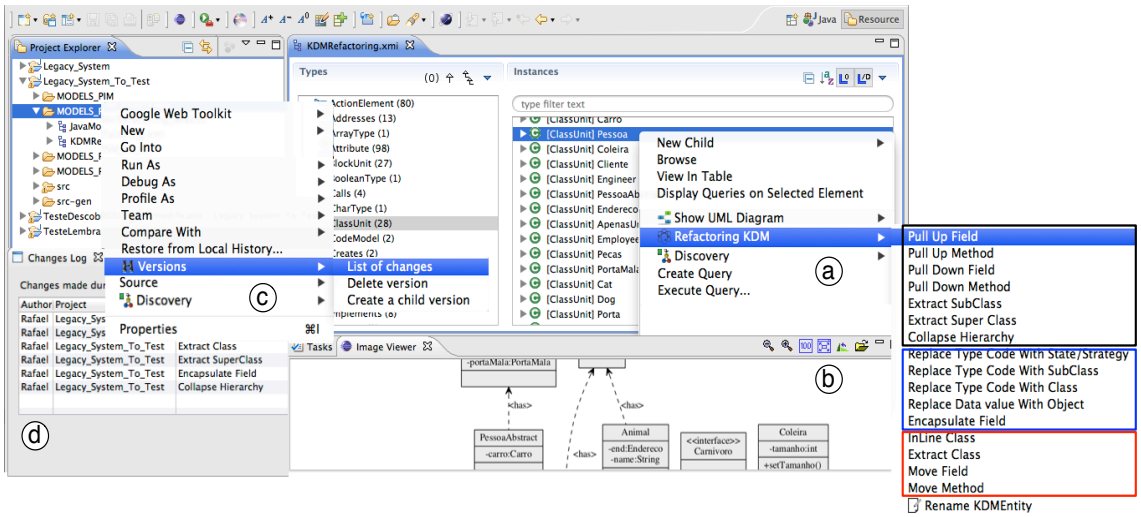

Figure 3. KDM-RE's Interface

before to apply some refactorings in order to assist the engineer to decide where to apply the refactorings or this UML class diagram can be generated as the engineer performs the refactorings in KDM model, i.e., changes are reproduced on the fly in a class diagram. We claim that the latter use of this UML diagram is important once the class diagram provides an abstract view of the system, hence, the engineer can visually check the system's changes after applying a set of refactorings. In addition, usually the source code is the only available artifact of the legacy software. Therefore, creating an UML class diagram makes, both the legacy software and the generated software to have a new type of artifact (i.e., UML class models), improving their documentation.

KDM-RE also supplies a multiple versions of a system at level models (KDM) which allows the engineer to work interactively on multiple models and to explore alternate refactoring path. As can see in the region ⓒ (see Figure 3), the engineer must select a KDM file, then he must right-click on the mouse to appear a popup menu named "Versions". By releasing the mouse on this menu, three options is shown: (*i*) List of changes, (*ii*) Delete version and (*iii*) Create a child version. The last option create a copy of the

KDM file - then the engineer can explore another refactoring path. The second option delete a specific version - first option shows all changes that have been realized in a current KDM file, all changes are depicted in an Eclipse View, as shown in region ⓓ. In this View it is possible to visualize the author that have committed the changes, the project and all refactorings realized.

2.2. Executing Refactoring KDM-RE

After the engineer clicks on the menu-item in region ⓐ upon Figure 3 and choose which refactoring to apply, a method `run()` in the class related to the chosen refactoring is being called. In this method the refactoring classes and a "RefactoringWizard" are started. In every refactoring the KDM file must be analyzed to find the meta-classes that are affected by a specific refactoring. Because of the structure of the KDM file, the easiest way to do this, is a traversal using the visitor pattern [Gamma et al., 1994]. Therefore we implemented the visitor pattern in KDM-RE to assist the travel of all meta-classes in a correct order.

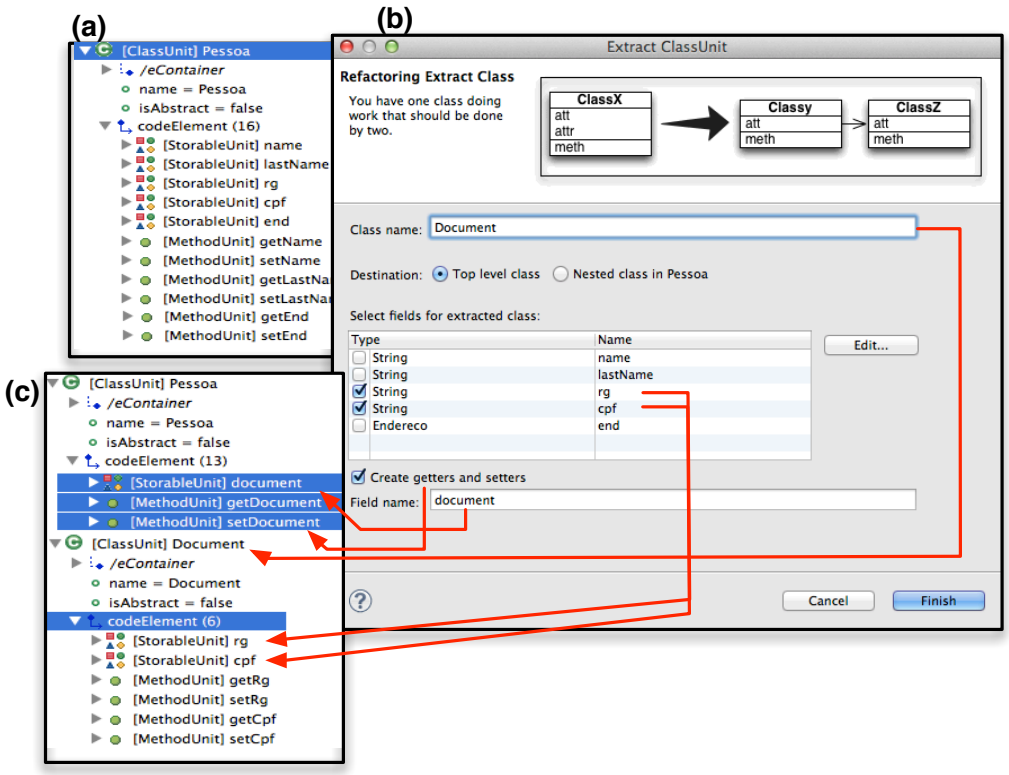

Figure 4. Extract Class Wizard

Notice that KDM-RE also provides a way to indicate refactoring opportunities. Refactoring itself will not bring the full benefits, if we do not understand when refactoring needs to be applied. Thus, to make it easier for the engineer to decide whether certain software needs refactoring or not KDM-RE implements a set of bad code smell according to [**?**]. For explanation purpose pretend that the KDM-RE found out that one class is doing work that should be done by two, thus, he must apply the refactoring "Extract Class". The first step is to select the metaclass that KDM-RE identified as a bad

smell, i.e., the metaclass to be extracted into a separate one, this step is illustrated in Figure 4(a). After selecting the metaclass, a right-click opens the context menu where the refactoring is accessible. After the click, the system displays the "RefactoringWizard" to the engineer, Figure 4(b) depicts the Extract Class Wizard. In this wizard, the name of the new metaclass can be set. Also a preview of all detected `StorableUnits` and `MethodUnits` that can be chosen to put into the new metaclass. Further, the engineer can select if either the new metaclass will be a top level metaclass or a nested metaclass. The engineer also can select if the KDM-RE must create instances of `MethodUnits` to represent accessors methods (gets and sets). Finally, the engineer can set the name of the `StorableUnit` that represent the link between the two metaclasses (the old metaclass and the new one). After all of the required inputs have been made, the engineer can click on the button "Finish" and the refactoring "Extract Class" is performed by KDM-RE. As can be seen in Figure 4(c) a new instance of `ClassUnit` named "Document" was created - two `StorableUnit` from "Pessoa", i.e., "rg" and "CPF" were moved to the new `ClassUnit` - instances of `MethodUnits` were also created to represent the gets and sets. In addition, the instance of `ClassUnit` named "Pessoa" owns a new instance of `StorableUnit` that represent the link between both `ClassUnits`. Due space limitation the other `StorableUnits` of `ClassUnit` named "Pessoa" are not shown in Figure 4(c).

After the engineer realize the refactoring an UML class diagram is created on the fly to mirror graphically all changes performed in the KDM model, see Figure 3ⓑ. Moreover, the KDM-RE creates/updates a tracking log to show the historic of all changes performed in the system, as can be see in Figure 3ⓓ.

2.3. Forward Engineering

After the engineer realize the refactoring the next step are to transform the KDM model to a PSM, i.e., a Java Meta-Model and to generate the refactored source-code conforming the PSM. The former step is carried out based on a set of transformations using ATL, due space limitation these transformations are not depicted. Then after transform the KDM to a instance of Java meta-model KDM-RE uses a textual template approach, such as the Acceleo[2] to regenerate the refactored source code. A template can be thought of as the target text with holes for variable parts, see Figure 5(b) - the holes contain metacode which is run at template instantiation time to compute the variable parts. After KDM-RE execute such template the refactored source-code is regenerated. In Figure 5(c) a chunk of the refactored source-code is depicted.

3. Architecture

In Figure 6 is depicted the architecture of KDM-RE which is split in three layers. As shown in this figure, the first layer of our *plug-in* is the Core Framework. This layer represents that we devised the *plug-in* on the top of the Eclipse Platform. In this layer it is also possible to see that we used both Java and Groovy as programming language. Moreover, this layer contains Eclipse Plugins on which our tool is based on, such as MoDisco and EMF. We used MoDisco[3] that is an extensible framework to develop model-driven tools

[2]http://www.eclipse.org/acceleo/

[3]http://www.eclipse.org/MoDisco/

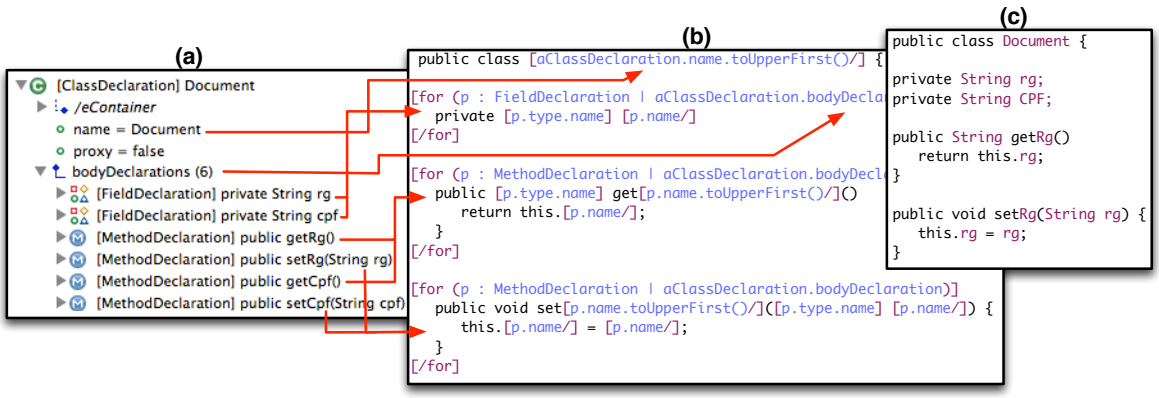

Figure 5. Forward Engineering steps

to support use-cases of existing software modernization and provides an Application Programming Interface - (API) to easily access the KDM model. Also, Eclipse Modeling Framewokr (EMF)[4] was used to load and navigate KDM models that were generated with MoDisco.

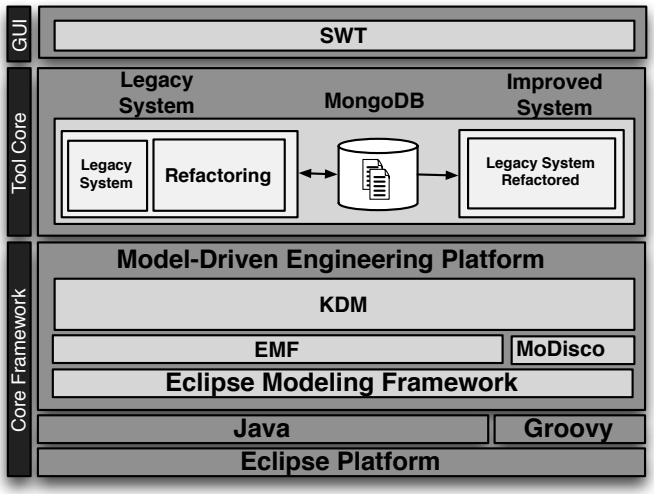

Figure 6. Architecture of KDM-RE

The second layer, the Tool Core, is where all refactorings provided by our *plug-in* were implemented. KDM-RE works intensively with KDM models, which are XML files. Therefore, we use Groovy to handle those types of files because of the simplicity of its syntax and fully integrated with Java. KDM-RE also provides a way to create multiple versions of the KDM file to allow the engineer to assess different refactorings in the same system. In order to optimizes memory usage of multiple versions for large models and enabling to work interactively on multiple models our *plug-in* persists these models in a MongoDB. We chose MongoDB since it provides a high performance. After the engineer to choose a version refactored a forward engineering is carried out and the

[4]http://www.eclipse.org/modeling/emf/

source code of the modernized target system is generated again. Finally, the top 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.

4. Related Work

Borger et al. [Boger and Sturm, 2002] developed a plug-in for the CASE tool ArgoUML that support UML model-based refactorings. The refactoring of class, states and activities is possible, allowing the user to apply refactorings that are not simple to apply at source code level. Van Gorp et al. [Gorp et al., 2003] proposed a UML profile to express pre and post conditions of source code refactorings using Object Constraint Language (OCL) constraints. The proposed profile allows that a CASE tool: (*i*) verify pre and post conditions for the composition of sequences of refactorings; and (*ii*) use the OCL consulting mechanism to detect bad smells such as crosscutting concerns.

The differential of our approach described herein in relation to the other is the proposal to move from software refactoring to model-driven refactoring by means of KDM, which is a platform and language independent metamodel.

5. Concluding Remarks

In this paper is presented the KDM-RE which provides support to model-driven refactoring based on ADM and uses the KDM standard. It follows the theory of the horseshoe modernization model, which is threefold: (*i*) Reverser Engineering, (*ii*) Refactoring and (*iii*) Forward Engineering.

Firstly, the engineer starts the refactoring process by means of a KDM file or a source code to be refactored. If the engineer provides a KDM file then KDM-RE can already apply the refactorings. Otherwise, the source code of the legacy system must be transformed into PSMs. Still in the first step, these PSMs are converted into a KDM model through a set of M2M transformation by means of MoDisco. Secondly, the engineer by using KDM-RE can apply a set refactorings in this KDM. Also, on the fly the engineer can check all changes realized in this KDM replicated into a class diagram - the engineer can visually verify the system‘s changes after applying a set of refactorings. KDM-RE also supplies a multiple versions of a system at level models. Finally, KDM-RE performs a forward engineering then a refactored source code is generated.

We believe that KDM-RE makes a contribution to the challenges of Software Engineering which focuses on mechanisms to support the automation of model-driven refactoring. Future work involves implementing more refactorings and conducting experiments to evaluate all refactorings provided by KDM-RE. Notice that KDM-RE is open source and it can be downloaded at *www.dc.ufscar.br/∼valter/KDM-RE*.

6. Acknowledgements

The authors would like to thank FAPESP for Process 2012/05168-4.

References

Boger, M. and Sturm, T. (2002). Tool-support for model-driven software engineering. In *In Proceedings of Practical UML-Based Rigorous Development Methods*.

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

Gorp, P. V., Stenten, H., Mens, T., and Demeyer, S. (2003). Towards automating source-consistent uml refactorings. In *International Conference on UML - The Unified Modeling Language*, pages 144–158. Springer.

Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois.