Title to be defined

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*—Refactorings are technique that assist developers in reformulating the overall structure of applications aiming to improve internal quality attributes. Due the growing interest in Model-Driven Reverse Engineering (MDRE) motivated the Object Manage Group (OMG) to launch the Architecture Driven Modernization (ADM) Task Force with the main objective to propose a set of standard meta-models for modernization projects. One of them is the Knowledge Discovery Metamodel (KDM), which contains meta-classes for representing the entire software system, such as: architecture, GUI, business processes, code, etc. However, although ADM provides the general concepts for conducting MDRE, it does not provide instructions on how to foster the reuse of refactorings in the KDM during the process of modernization. This leads developers to create their own refactoring solutions, resulting in a delay in the modernization. In order to fill this gap, we present a novel meta-model to foster the reuse of refactorings by using the KDM. We discuss how refactorings can be created and further reused using our meta-model. To evaluate our meta-model we carried out an experiment using. The results showed that our meta-model can be used to reuse refactoring...**

We have also developed a domain specific language to support the instantiation of our metamodel and making it easier to foster the reuse of refactorings.

I. Introduction

In 2003 the Object Management Group (OMG) created a group called Architecture-Driven Modernization Task Force (ADMTF). Its goal was to analyze and evolve conventional reengineering processes, formalizing them and making them to be supported by models [**?**]. ADM advocates the conduction of reengineering processes following the Model-Driven Architecture (MDA) principles [**?**] [**?**], i.e., all the main software artifacts considered along with the process are Platform-Independent Model (PIM), Platform-Specific Model (PSM) or Computational Independent Model (CIM).

According to OMG the most important artifact provided by Architecture-Driven Modernization (ADM) is the Knowledge Discovery Meta-model (KDM). By means of KDM is possible to represent all system's artifacts, such as configuration files, graphical user interface, architectural views and source-code. It is divided into four layers, representing physical and logical software assets at different abstraction levels. Each layer is further organized into packages that, by their turn, define a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to legacy systems. One of the primary uses of KDM is in reverse engineering, in that a parser reads the source-code of systems and generates KDM instances representing them. After that, refactorings and otimizations can be performed over this model, aiming to solve previously identified problems (**colocarNossasREFERENCIAS**). The main idea behind KDM is that the community starts to create approaches and tools that work over KDM instances - thus, every approaches/-tools that use KDM as input can be considered platform and language-independent. For instance, a catalog of refactoring, a concern mining technique, an aspect oriented modernization, all of these approaches if they use KDM as input can be considered platform and language-independent.

Refactoring can improve maintainability and reusability of systems [**?**]. Not only existing research which suggests that refactoring is useful, but it also suggests that refactoring is a frequent practice [**?**]. However, although ADM, and mainly KDM, have been proposed to support the modernization of legacy systems, up to this moment, the KDM does not contains meta-classes suitable for fostering the reuse of refactorings. Therefore, software modernizers/reengineers need to develop their own solutions; usually these solutions are proprietary and very difficult to reuse.

In order to overcome the aforementioned limitation, in this paper we present a novel refactoring meta-model supported by KDM. Our main reasons to create this meta-model are:

- fostering the reuse of refactoring is extremely important as refactoring is a bounded, well defined set of disciplines that are well understood and provide value today;
- addressing refactoring as a separate meta-model opens up options to project teams and to tool vendors as to how and when they wish to refactor applications vs. transform those applications to new target architectures.

Therefore, we claim that it is important to provide a way to promote reuse of refactorings - this can be done by means of our refactoring meta-model. Moreover, in order to provide some evidence of the our dedicated refactoring catalogue for KDM we performed an experiment using eight open source Java application. More specifically, we conducted a reverse engineering in order to get the KDM model of these eight open source Java application. Then, we applied three different refactorings in their KDM models. Experimental results show that the our dedicated refactoring catalogue improved the

legacy system's cohesion.

Therefore, the main contributions of this paper are threefold: (*i*) we show a dedicated refactoring catalogue for KDM; (*ii*) we demonstrate the feasibility of our catalogue by implementing it as an Eclipse plug-in; *(iii)* we show the results of an experiment by applying some refactorings to eight open source programs and their KDM models.

II. Background

A. Architecture-Driven Modernization

The growing interest in Model-Driven Reverse Engineering (MDRE) motivated the OMG to launch the ADM whose goal was to establish standards for reengineering processes [**?**]. This initiative was motivated by considering a high number of unsuccessful reengineering projects reported by literature and companies [**?**], [**?**], [**?**]. As a result of this effort it was created the ADM concept, aiming at revitalizing existing software systems just by using standard models and employing well known MDA principles. ADM solves the formalization problem since it represents all the artifacts involved in the reengineering process as standard models [**?**].

According to OMG [**?**] the ADM goal is not to replace the traditional reengineering processes, but improves them through the use of MDA. ADM consists in an adaptation of the well-known horse-shoe model. One of the main aim of ADM is not only follows the principles of MDA, ADM has also set off the development of a set of standards to deal with different challenges that appear in the modernization of legacy information systems [**?**]. In this context, the ADMTF issued the request-for-proposal of Knowledge Discovery Meta-Model (KDM). More information upon KDM are presented as follow.

B. Knowledge Discovery Metamodel (KDM)

KDM is a meta-model for representing existing software, its elements, associations, and operational environments. According to [**?**] the KDM goals is to be a meta-model to allow software engineer to create tools to exchange application metadata across different application, languages, platforms and environments. KDM is an OMG discovery meta-model specification; and nowadays it is being adopted as *ISO/IEC 19506* by the *International Standards Organization* for representing information related to existing software systems.

The KDM consists of four abstraction layers. In this paper we are specially interested in the `Program Elements Layer` which contains the `Code` and `Action` packages. These packages collectively define the `Code model` that represents the implementation level assets of the existing software system, determined by the programming languages used in the developments of the existing software system.

The `Code` package defines a set of meta-model elements whose purpose is to represent implementation level program elements and their associations. It includes meta-classes, which represent common program elements supported by various programming languages, such as: classes, procedures, macros, prototypes, and templates.

In order to fully understand how KDM is used to represent the source code of a specific program, in Listing 1 is shown a simplified example in Java. The corresponding, though simplified KDM instance is depicted in Figure 1. It illustrates a KDM instance as a UML object diagram for the sake of simplicity, note that this diagram represents the source code as a tree of nodes containing some KDM's meta-classes. As can be seen in Figure 1 the root meta-class is `Segment`, which is a container for a meaningful set of facts about an existing software system. Each `Segment` may include one or more KDM model instances, such as `CodeModel`.

Listing 1: A Simply Java Source-code.

```
❶ package model;
❷ public class Car ❸ extends ❹ Vehicle{
➥❺ private String name;
➥❻ public String getName(){
    ...
  }
}
```

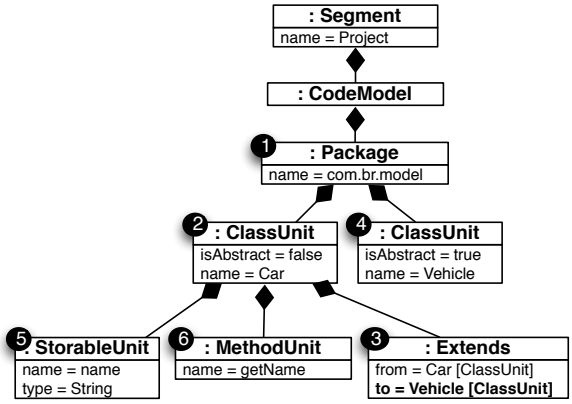

Fig. 1: A corresponding KDM instance of Listing 1

Analyzing both the Listing 1 and the Figure 1 it is evident that each static structure of the source code has a meta-class in KDM to represent it. For instance, the package model in line 1 of Listing 1 ❶ is represented in KDM by the meta-class named `Package`, see Figure 1 ❶. Next, the class `Car` is declared, see Listing 1 ❷. It also inherit from class `Vehicle`, in the Java this is accomplished by using the keyword `extends` following of a class, see Listing 1 ❸ and ❹. In KDM classes are represented by the meta-class `ClassUnit` as depicted in Figure 1 ❷ and ❹. The meta-class `Extends` represents inheritance in KDM models. As shown in Figure 1 ❸ the meta-class `Extends` has two association, `to` and `from`, the former represents the parent class (super class), and the latter represents the child class (sub-class). In this context, the child is the class `Car` and the parent class is `Vehicle`, which is depicted in Figure 1 ❹.

Finally, the variable `name`, and the method `getName()` (see Listing 1 ❺, and ❻) are mapped to corresponding instances of the KDM elements, `StorableUnit`, and `MethodUnit` (see Figure 1 ❺, and ❻).

III. Refactoring Meta-model

Suppose the following scenario, an engineer defines a specific refactoring and share it through a repository - other engineers can then browser this refactoring, download and reuse it. This could be accomplished by defining a meta-model. Our goal is to create a meta-model that allows representing both fine-grained as macro refactorings, but still respecting the language and platform independence offered by KDM. One important characteristic of this meta-model is that it need to work along with all KDM meta-classes - it can be easily incorporated in existing KDM tools.

Addressing refactoring as a meta-model can open up options to foster the reuse of refactorings. Engineers could use this meta-model to specify and reuse refactorings. In this context, this section describes a possible refactoring meta-model that we have devised. This refactoring meta-model can also be seen an extensible specification for exchanging refactoring information.

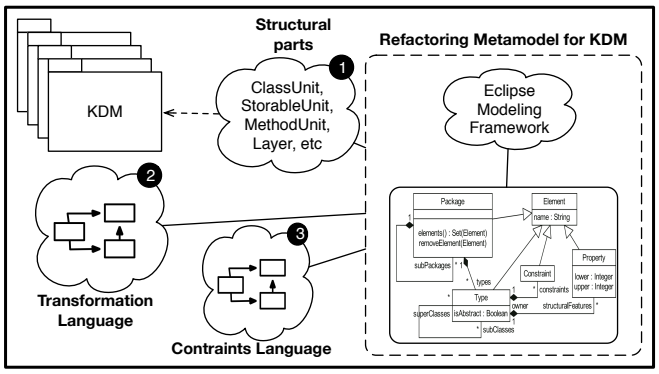

Fig. 2: Overall approach for creating refactoring meta-model

In order to foster the reuse of refactorings for KDM in previously work we adapted some fowler's refactorings to KDM [**?**]. This was fundamental to identify potential for reuse of refactoring and for building the refactoring meta-model. As result, we identified that reusing refactorings for KDM requires a combination of three key elements, which are visible in Figure 2. As can be seen in this figure, firstly structural parts of a refactoring were considered as a good candidate to reuse, i.e., the elements that are transformed. Consider for example the basic refactoring *RenameX*, where *X* can be any elements. The steps needed to achieve this refactoring are equal no matter what kind of element needs to be renamed. From this simple example we gained some initial insights. As KDM was built to represent the whole system and its associations - it is a good candidate to represent the structural parts of a refactoring.

Further, we could also identify that every refactoring has at least more two parts. The transformational one, i.e., the refactoring's operations, which is usually performed by a Model-to-Model transformation language (either Query/View/-Transformation (QVT) or ATL Transformation Language) - and a set of constraints, which are pre- and postconditions to check the correctness of the applied refactoring, i.e., usually refactorings pre- and postconditions are written in Object Constraint Language (OCL).

Combining these three elements (see Figure 2) we have an initial insight about a possible refactoring meta-model. The meta-model is shown in Figure 3. Consistent with other models defined by OMG, our refactoring meta-model is defined using the MOF meta-modeling language. As such, it has a standard textual representation presented by XMI. We have modeled it this way in order to quickly generate model editors for validation purposes. This meta-model is a set of enumerations and meta-classes, which are either concrete or abstract. The meta-classes of which our meta-model consists are defined as follows.

- `RefactoringModel` it is used to hold information about a refactoring.
  - **Associations**
    - `author:Author[0..1]`: specifies the refactoring's author;
    - `libraries:RefactoringLibrary[0..*]`: the set of all `RefactoringLibrary` owned by the `RefactoringModel`.
- `Author` represents the author of the refactoring. It provides two meta-attributes.
  - **Attributes**
    - `name`: specifies the author's name;
    - `lastName`: defines the author's last name.
- `RefactoringLibrary` represents the top container for all refactorings.
  - **Attributes**
    - `name`: specifies the name of the refactoring library;
    - `shortDescription`: a short description for the refactoring library;
    - `description`: a detailed description for the refactoring library.
  - **Association**
    - `catalogs:Catalog[0..*]`: a set of catalogs that contains refactorings.
- `Catalog` is used to model the catalog of refactorings.
  - **Attributes**
    - `name`: defines the catalog's name.
  - **Association**
    - `author:Author[0..1]`: specifies the creator of catalog;
    - `refactorings:Refactoring[0..*]`: the set of all `Refactoring` owned by the `Catalog`.
- `Refactoring` represents the main meta-class in our meta-model.
  - **Attributes**
    - `name`: identifies the refactoring and helps to build a common vocabulary for software developers;

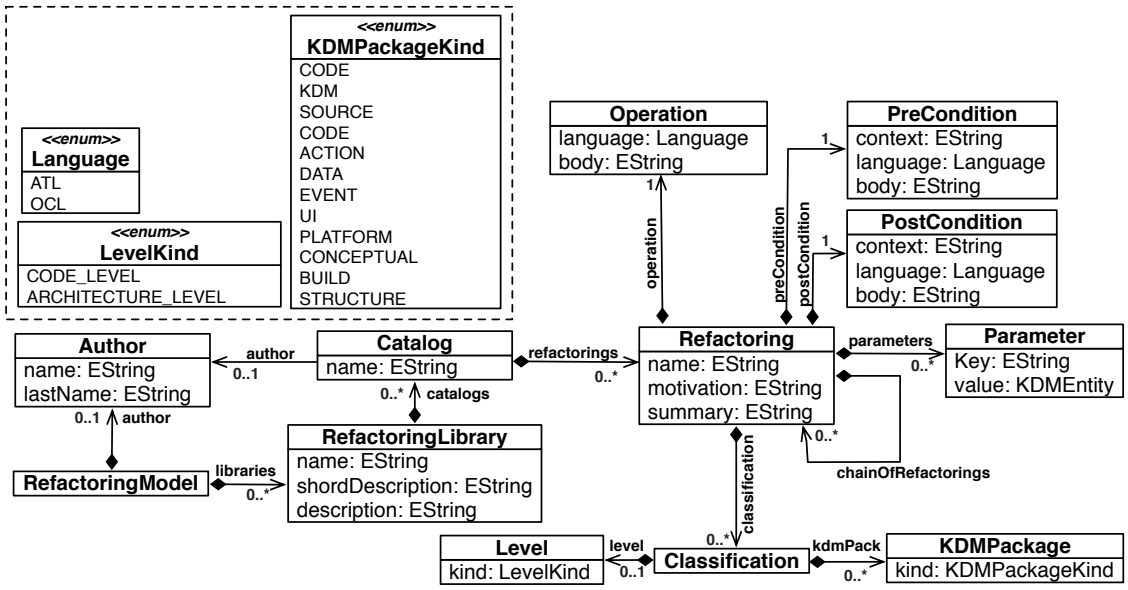

Fig. 3: Refactoring Meta-model.

* `motivation`: describes why the refactoring should be done and lists circumstances in which it should not be used;
* `summary`: tells when and where you need the refactoring and what it does. It also helps engineer to find a relevant refactoring in a given situation.

– **Association**

* `operation:Operation[1]`: defines an operation to execute;
* `preCondition:PreCondition[1]`: a precondition that need to be satisfied before to apply the refactoring;
* `postCondition:PostCondition[1]`: a postcondition describes the effect of applying a refactoring ;
* `parameters:Parameter[0..*]`: a set of parameters that are used to perform the refactoring;
* `chainOfRefactoring:Refactoring[0..*]`: a set of refactorings that as combined can perform more complex refactorings;
* `classification:Classification[0..*]`: defines the classification of the refactoring.

- `Operation` defines an operation to execute, i.e., the mechanic that provides a step-by-step description of how the refactoring is performed.
  – **Attributes**
  * `language`: Specifies the language of the operation. Valid values are currently "ATL" and "QVT";
  * `body`: Specifies the refactoring operation expressed in the selected language.
- `PreCondition` defines a precondition to execute before to execute the refactoring.
  – **Attributes**
  * `context`: Specifies the classifier for which this helper is being defined. OCL inheritance rules applies to resolve applicability of operation, based on the passed in context;
  * `language`: Specifies the language of the operation. Valid value is currently "OCL";
  * `body`: Specifies the body of the OCL helper method.
- `PostCondition` defines a postcondition to execute after to execute the refactoring.
  – **Attributes**
  * `context`: Specifies the classifier for which this helper is being defined. OCL inheritance rules applies to resolve applicability of operation, based on the passed in context;
  * `language`: Specifies the language of the operation. Valid value is currently "OCL";
  * `body`: Specifies the body of the OCL helper method.
- `Parameter` defines a set of parameter needed to execute the refactoring. We used a structure that can map keys to values, similar to hash table.
  – **Attributes**
  * `key`: represent the name of the parameter;
  * `value`: represents the type of the parameter. The value must be either primitives types or KDM's meta-classes.
- `Classification` defines the classification of the refactoring.
  – **Association**

    * `level`: represent if the refactoring is either low-level refactoring or high-level refactoring;
    * `kdmPack`: defines which kdm's packages is needed to execute the operation of a refactoring.

- `Level` used to defines the level of the refactoring, i.e., if the refactoring is either low-level refactoring or high-level refactoring.
  - **Attributes**
    * `kind`: specifies some common properties of the `Level`.
- `KDMPackage` which kdm's packages is needed to execute the operation of a refactoring.
  - **Attributes**
    * `kind`: specifies some common properties of the `KDMPackage`.

Three enumeration also have been defined, illustrated in the portion of Figure 3 bounded by a dashed line shape. The first one is `Language` that is used to specify the language of the operation, valid values are currently "ATL" and "OCL". The second enumeration is `LevelKind` used to define if the refactoring is either low-level refactoring or high-level refactoring. Finally, the last enumeration is `KDMPackageKind` - used to define which KDM's packages are used during the executing of a refactoring.

A. Specifying Refactorings

In this section, we briefly show an example of how to use the meta-model described in Section III to promote the reuse of refactorings. Meta-models can be serialized and deserialized without the loss of information, i.e., they are represented as XML Metadata Interchange (XMI). Therefore, to foster the reuse of refactoring, firstly the engineer must instantiate the refactoring meta-model.

This step is very important since other engineers will use such information to search a specific refactoring. As previously highlighted our meta-model was developed using the Ecore API. However, the use of this API to instantiate the meta-classes is an error-prone activity as the coding is very verbose and time consuming. To increase the level of abstraction it is often required to design the representation in a way that it becomes specific to a certain domain. This basically means to leave out detail and expressiveness that is not needed for this field of application and to introduce constructs that are specialized to this domain. The refactoring's meta-model representation has to be readable, and ideally, also writable by the engineer it is designed for. To fulfill this need we have devised two environments to facilitate the instantiation of our refactoring's meta-model.

The first one is a Domain-Specific Language (DSL). The DSL has been devised on top of Eclipse technology - we "invert" the parser and match the grammar rules (which are the basis for the parser) against the refactoring's meta-model and thereby recreate the textual representation. This process is supported by XText[1]. The second environment is a web application in which engineers can make available refactorings and/or can browser other refactorings already available. Due space limitation only the DSL is presented herein. Nevertheless the web application can be reached at www.reuseRefactoring.com . Further details of these environment are out of the scope of this paper.

In Figure 4 is shown a chunk of code written in the aforementioned DSL. The boxes highlighted in dark gray represent specific parts of DSL that deserve attention: (*i*) the refactoring's operation written in ATL code, and (*ii*) the refactoring's precondition written in OCL.

```
1 refactoringModel {
...
4 refactoringLibraries {
5  name : FineGrainedRefactoring
6  shortDescription : "contains a set of refactorings"
7  description : "refactorings"
8  catalog {
9   name : FowlersRefactoring
10   author : Rafael
11   refactoring {
12    name : renameStorableUnit
13    motivation : "The name of a StorableUnit does not reveal its purpose."
14    summary : "Change the name of the StorableUnit."
15    operation {
16     language : ATL
17     body : {
18   "-- @atlcompiler atl2010
19   -- @nsURI MM=http://www.eclipse.org/MoDisco/kdm/code
20   module renameAttribute;
21   create OUT : MM refining IN : MM;
22    rule renameAttribute {
23      from
24       s : MM!StorableUnit (thisModule.getClassUnitContainer(s).name = '#parameter0'
25        and s.name = '#parameter1')
26      using {
27       newName : String = '#parameter2';
28    }
29      to
30      t : MM!StorableUnit (
31       name <- newName)
32    }
33   helper def : getClassUnitContainer (attribute : MM!StorableUnit) :
34    MM!ClassUnit = attribute.refImmediateComposite();"
35  }
36  }
37   preCondition {
38    context : "StorableUnit"
39    language : OCL
40    body : {
41     "#parameter1 <> #parameter2 and c.allConflictingNames()->excludes(#parameter2)"
42    }
...
}
```

Fig. 4: Chunk of code used to instantiate the refactoring meta-model.

Each word in **bold** represents DSL keywords- each keyword match the DSL's grammar rules against the refactoring's meta-model, i.e., its meta-classes and its meta-attributes. For instance, the meta-class `RefactoringModel` presented in Figure 3 is declared in line 1 (see Figure 4) by using the keyword **refactoringModel**. The **refactoringModel** body (the area between the braces) holds information about the refactoring.

[1]Xtext is an Eclipse framework for implementing programming languages and DSLs. It lets you implement languages quickly, and most of all, it covers all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a complete Eclipse IDE integration

After the first declaration, it is mandatory to declare the author (not depicted in Figure 4) and the refactoring libraries. The keyword **refactoringLibraries**, depicted in line 4, represents the instantiation of `RefactoringLibrary` - lines 5, 6, and 7 set its meta-attributes' value, **name**, **shortDescription**, and **description**, respectively.

In line 8 the meta-class `Catalog` is declared. This catalog has the **name** "FowlersRefactoring" (see line 9) and in line 10 the **author** is defined. Line 11 the meta-class `Refactoring` is defined. This implies that its meta-attributes also need to be defined, i.e., **name**, **motivation**, and **summary** are defined in line 12 through 14, respectively. Furthermore, in line 15 the meta-class `Operation` is declared using the keyword **operation** followed by curly braces. In line 16 the operation's **language** is specified. In this example we have chosen ATL as the transformation language[2], but we could choose QVT.

In line 17 to 34 the refactoring's operation is defined using the keyword **body**. The underlined words illustrate the parameters that should be overwritten during the reuse of this refactoring's operation. This operation contains one rule responsible of renaming a `StorableUnit`. Occurrences of the input pattern may be filtered by introducing a guard, a boolean condition that source model elements must satisfy, e.g. lines 24 and 25 identify the correct `StorableUnit` and check if it is contained in the correct `ClassUnit`. Lines 26 to 28 a variable is declared. Line 31, the element of the output, e.g., the `StorableUnit` has it meta-attribute `name` renamed. Lines 33 and 34 depicts a helper, which can be viewed as the ATL equivalent to methods. It returns the contained `ClassUnit` of a specific `StorableUnit`

In line 37 the meta-class `PreCondition` is instantiated using the keyword **preCondition**. In lines 38 and 39 its meta-attributes are defined. Then in line 40 to 42 its meta-attribute **body** is specified using OCL. This OCL code verifies if the **#parameter1** is different from **#parameter2**, i.e., checks if the `StorableUnit` does not already contain the name to be changed. Note that to reuse the refactoring's operations as well its pre- postconditions all parameters need to be overwritten, see Subsection III-B.

After completing the instantiation of refactoring's meta-model using the DSL, it is possible to execute it and generate a serialized file, a XMI file. Further, this XMI file is persisted either into hard disk or to send it over a repository. If the XMI file is send to a repository it can be browsed and reused using the our web application. Figure 5 depicts a snippet of this XMI file.

Information upon how to reuse the instantiated refactoring's meta-model is presented as follows.

B. Reusing Refactoring

In this step is described how to reuse all refactoring's information that were serialized. Therefore, as a usage scenario suppose that the refactoring (the XMI file) that was instantiated

[2]ATL is one of the most widely used transformation languages, both in academia and industry, and there is mature tool support available.

```
<?xml version="1.0" encoding="ASCII"?>
<refactoringModel:RefactoringModel xmi:version="2.0" ">
 <libraries name="FineGrainedRefactoring" shortDescription="contains a set of
 refactorings" description="refactorings">
  <catalogs name="FowlersRefactoring" author="//@author">
   <refactorings name="renameStorableUnit" motivation="The name of an attribute
   does not reveal its purpose." summary="Change the name of the StorableUnit."
    <preCondition context="StorableUnit" language="OCL" body="..."/>
    <parameters xsi:type="code:ClassUnit" name="Client" kind="local"/>
    <parameters xsi:type="code:StorableUnit" name="rg" kind="local"/>
    <parameters xsi:type="code:String" name="CPF" kind="local"/>
    <operation language="OCL" body="...."/>
   </refactorings>
  </catalogs>
 </libraries>
 <author name="Rafael" lastName="Durelli"/>
</refactoringModel:RefactoringModel>
```

Fig. 5: A snippet of the refactoring meta-model instantiated in XMI format.

earlier has been sent to a repository. Afterwards an engineer browsed this repository and chose the refactoring to apply to his system. However, before to apply the refactoring, it is important to highlight that the system that will be refactored need to be transformed into a KDM instance. This can be accomplished easily by using MoDisco[3], which has a parser that automatically transforms Java source code into KDM XMI instances.

Next, to reuse the refactoring the engineer must download of the XMI file (the refactoring). Then, the engineer must use/adapt the informations persisted in this XMI file, i.e., this XMI file contains all informations necessary to perform the refactoring (see Figure 5), such as: the refactoring explanation, operation, pre- postcondition, etc.

To reuse the refactoring's operations as well its pre- postconditions all parameters need to be overwritten. However, change all parameters (see Figure 4) of a refactoring can be a costly and burdensome activity - imagine an operation with dozens of parameters to be changed. In order to fulfill this need we devised a Eclipse plugin. It assists the engineer to reuse and adapt the refactoring's operation parameters for his context. This plugin uses as a starting point two artifacts: (*i*) an instance of KDM that represents the system to be refactored, and (*ii*) and the downloaded XMI file, which represents an instance of the refactoring meta-model. The former artifact is used to gather information about the system to be refactored, e.g., the plugin seeks all instance of `ClassUnit`, `MethodUnit`, `StorableUnit`, etc - these instances can be used as parameter values to perform the refactoring. The later artifact is used to obtain the refactoring's operation and identify all the parameters that need to be replaced to "real" values.The main screenshot of the plugins is shown in Figure 6.

This screenshot is used to overwrite all refactoring's parameters. Note that all parameters illustrated in Figure 4 lines 18 to 34 are presented also in Figure 6. By means of this screenshot it is possible to choose and overwrite the values for each parameter. This plugin allows one to choose

[3]https://eclipse.org/MoDisco/

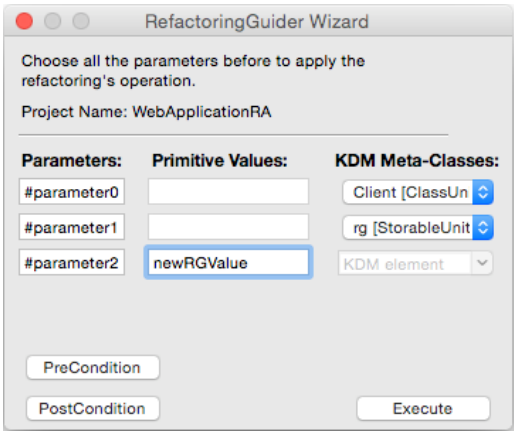

Fig. 6: Parameters Editor.

either primitive values (string, int, double, etc) or KDM meta-classes (`ClassUnit`, `MethodUnit`, `StorableUnit`, etc). As can be seen in Figure 6 the **#parameter0** has been set to an instance of `ClassUnit`. Similarly, the **#parameter1** has been set to an instance of `StorableUnit`, and finally the **#parameter2** has been set as a string. After completing the parameters one should click in execute - then if the refactoring's operation will be run and an refactored KDM instance will appear; in this instance, the refactored KDM is available.

IV. Related Work

The approach proposed by Cechticky *et al.* [1] 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.* [2]. 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.* [1] 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.

V. Conclusions

In this paper, a model-based process was presented, which raises abstraction levels of CF reuse. It serves as a graphical view that replaces textual cookbooks and is used to perform the reuse in a model driven approach. From our proposed model-based approach, a new reuse process was delineated, which employs the forms during the development of a new application, allowing engineers to start the reuse since earlier software development phases and reduce the time to reuse a CF. With this, application developers do not need to worry about reuse coding issues nor how the framework was implemented, allowing to focus on the reuse requirements in a higher abstraction level.

Our approach was evaluated in two experiments that could answer the questions of the study planning, which indicate their conclusive success. The links for the gathered data can be accessed on *http://www2.dc.ufscar.br/~valter/*. The results regarding the productivity of reuse process were promising. However, the results of the maintenance study showed that our technique has no disadvantages in maintenance effort.

Furthermore, we have identified some limitations related to our research project. Once the models have been devised on top of the Eclipse Modeling Project, they can not be used in another environment. Furthermore, the code generator only generates code for Java and AspectJ, therefore, only frameworks developed in these languages are currently supported.

It is also important to point that our tool is part of a project to develop an integrated development environment for CF, which currently supports CF feature subset selection and a CF repository service. It is important to note that our tool also supports CFs that do not employ feature selection, in these cases, the RRM and RMs would be exactly equal.

However, we have not yet evaluated how to deal with coupling multiple CFs to a single base application. Despite this functionality already being supported, some frameworks may conflict with each other and lead to unwanted results.

The code generated is based on AspectJ and it was not evaluated if it supports every CF without modifications. Although not stated, we have also worked on selecting subsets of features of the framework.

Long term future works regard: (*i*) carry out a experiment using other CF, for verifying if the models proposed assist both the reuse and to maintain a reuse code; (*ii*) execute a experiment to verify whether the abstraction of the elements related to the models are sufficiently ideal; (*iii*) evaluate the standpoint of the domain engineers/frameworks, (*iv*) improve the elements of the models, i.e., better them graphically

and (*v*) analyze the reusability of the abstract metamodel's metaclasses.

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] 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

[2] 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