KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel

Bruno M. Santos, Raphael R. Honda,
Valter V. de Camargo
Departamento de Computação
Universidade Federal de São Carlos - UFSCar
São Carlos – SP – Brazil
{bruno.santos, raphael.honda, valter}@dc.ufscar.br

Rafael S. Durelli
Instituto de Ciências Matemáticas e Computação
Universidade de São Paulo - USP
São Carlos – SP – Brazil
rsdurelli@icmc.usp.br

***Abstract*— Architecture-Driven Modernization is the new generation of software reengineering. The main idea is to modernize legacy systems using a set of standard models. The first step is to obtain, by reverse engineering, an instance of an ISO metamodel called KDM that represents all details of the legacy system. Then, refactorings and optimizations can be applied over this model turning it into a target/modernized KDM. Afterwards the source code of the target system can be generated. In its original form, KDM does not provide aspectual concepts, preventing an aspect-oriented modernization to be properly conducted. In this paper we present KDM-AO, an aspect-oriented heavyweight extension for the KDM metamodel. The extension has been created based on a well known aspect-oriented profile for AspectJ language. To evaluate our extension, we applied it in an aspect-oriented modernization whose goal was to remodularize the persistence concern of an application using a Persistence Crosscutting Framework. The case study showed that KDM-AO is able to represent high-level and low-level aspect-oriented abstractions.**

***Keywords*— KDM profile; Architecture-Driven Modernization; KDM; aspect-oriented modernization; Crosscutting Frameworks**

I. INTRODUCTION

Systems are termed as "legacy" when their maintenance and evolution cost increasingly rise to unbearable levels, but they still deliver great and valuable benefits for companies. In order to make information systems continue satisfying their previously established requirements, they need to be continuously evolved or they probably will fail in fulfilling their goals. Many companies have systems that suffer the phenomena of erosion and aging. These phenomena are result of successive changes systems suffer along years of maintenance, for example, functionalities that were removed, modified or added; hence compromising their overall quality [1][4].

In 2003 the Object Management Group (OMG) created a task force called Architecture Driven Modernization Task Force (ADMTF). It was aimed to analyze and evolve typical reengineering processes, formalizing them and making them supported by models [2]. ADM advocates the conduction of reengineering processes following the principles of Model-Driven Architecture (MDA) [22][2], i.e., all the 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 using KDM, it is possible to represent all system's artifacts, such as configuration files, graphical user interfaces, architectural views and source-code details. The idea behind KDM is to motivate the community to start creating parsers and tools that work over KDM instances; thus, every tool that takes KDM as input can be considered platform and language-independent. For instance, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages [32]. One of the primary uses of KDM is during reverse engineering processes, in that a parser reads source-code of a system and generates a KDM instance representing it. After that, refactorings and otimizations can be performed over this model, aiming to solve previously identified problems.

Whenever one decides to modernize legacy systems aiming to remodularize concerns, a candidate paradigm is Aspect-Orientation (AO), which provides abstractions to improve the modularization of crosscutting concerns [29]. Although ADM/KDM had been created to support modernization of legacy systems, the original version of the KDM does not contains metaclasses suitable for representing AOP concepts [32]; hampering modernization processes whose goal is to remodularize crosscutting concerns [3].

A possible alternative is to extend KDM using a lightweight solution (Profiles) that is based on set of stereotypes and tag definitions. Profiles are able to impose restrictions on existing metaclasses, respecting the metamodel. However, the lightweight extension mechanism provided by KDM does not guarantee type checking in models, transferring all that responsibility for tools and Software Engineers.

In order to overcome the aforementioned limitation, in this paper we present a heavyweight extension for KDM called KDM-AO. Heavyweight extensions are based on a modified KDM metamodel, including new metaclasses or changing the existing ones. The goal was to create an extension that allows representing both high-level as low-level details, but still respecting the language and platform independence offered by KDM. One important characteristic of our heavyweight extension is that we have not changed the existing KDM metaclasses, we had just added new ones. Therefore, it can be easily incorporated in existing KDM tools.

978-1-4799-4223-7/14 $31.00 © 2014 IEEE
DOI 10.1109/SBES.2014.20

Our KDM-AO is totally based on an existing UML profile for creating class diagrams with AO stereotypes proposed by Evermann [8]. However, although Evermann´s profile is specific to class diagrams, when its stereotypes are mapped to the KDM, the KDM extension inherits all infrastructure available for this metamodel, allowing one to represent all static and dynamic details of a system. To support the creation of KDM-AO instances, we have also created an Eclipse plug-in to facilitate this process.

Another contribution of this paper is to present a preliminary mapping between UML metamodel and KDM metamodel. This mapping is a conceptual tool for converting UML profiles in KDM extensions. The success of modernization processes is heavily dependent on the abstractions which are possible to be represented in KDM. As most of the abstractions of recent domains (web services, embedded systems, aspects, business processes, cloud, etc) are not presented in KDM in an explicit way, we consider the conversion of UML profiles in KDM extensions (either heavy or lightweight) an important activity.

In order to assess our KDM-AO we carried out a Crosscutting Framework-based modernization process in a management system of a CD Shop [3]. The evaluation showed that KDM-AO is able to represent all the details inherent in this type of framework, as well as all AO concepts. In addition, the results show that by using the KDM-AO is possible to modernize a legacy system to AOP. However, it is beyond our scope the forward engineering of the system.

This paper is structured as follows: Section II shows background about ADM/KDM and Aspect-Oriented KDM. In Section III the Aspect-Oriented KDM is described. A case study is shown in Section IV. The related works are shown in Section V. Finally, in Section VI, the discussions and conclusions are presented.

II. BACKGROUND

A. ADM/KDM

In 2003, OMG initiated efforts to standardize the process of modernization of legacy systems using models by means of the ADMTF [2]. The aim of the ADM is the revitalization of existing applications by adding or improving functionalities, using existing OMG modeling standards and also considering MDA principles. In other words, the OMG through ADMTF took the initiative to standardize reengineering processes.

According to ADM [2], ADM does not replace reengineering, but improves it through the use of MDA. The basic process flow of modernization has three phases: Reverse engineering, restructuring and forward engineering. In the reverse engineering, the knowledge is extracted and a Platform-Specific Model (PSM) is generated. The PSM model serves as the basis for the generation of a Platform Independent Model (PIM) called KDM. Then this PIM can serve as basis for the creating of a Computing Independent Model [2].

In order to support the modernization process, in 2006 the KDM metamodel was created. It can be used to represent the system and their operating environments. KDM is language and platform-independent, i.e., a PIM that is able to represent physical and logical artifacts of legacy systems at different levels of abstraction. 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 [2]. These layers are created automatically, semi-automatically or manually through the application of various techniques of extraction of knowledge, analysis and transformations [5]. Fig. 1 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 [2].

Herein, we are especially interested in the Program Elements Layer because it defines the Code package which is widely used by our extension. The Code package possesses a set of metaclasses to represent program elements in implementation level. In other words, this package contains a set of metaclasses that represent the common named elements in the source code supported by different programming languages such as data types, classes, procedures, methods, templates and interfaces [6].

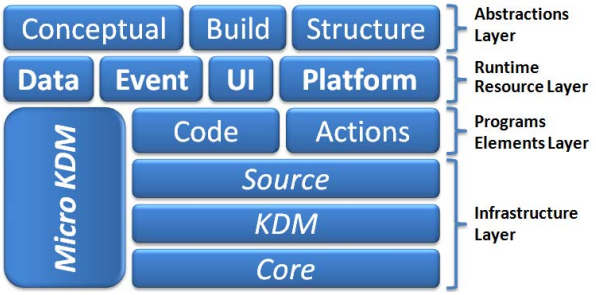

Figure 1. KDM Architecture [2] (Adapted).

As in UML, it is also possible to define either lightweight or heavyweight extension in KDM by means of extension mechanism. Heavyweight extensions are based on a modified KDM metamodel, including new metaclasses or changing the existing ones. On the other hand, lightweight extensions (also known as profiles) are based on set of stereotypes, tag definitions, and constraints, which are basically "notes" over the model. Profiles are able to impose restrictions on existing metaclasses, but they respect the metamodel, without modifying the original semantics of the elements. One of major benefits of profiles is that they can be handled in a natural way by existing tools.

In general, the drawback of heavyweight extensions is that existing tools get no longer compatible with the new metamodel. However, the only way to guarantee model correctness in model level is using heavyweight extensions. This happens because it is possible to relate metamodel elements by their types and not just by their names, as it usually happens in lightweight extensions. Using lightweight extensions, the correctness of the model must be guaranteed by tools. Besides, when heavyweight extensions do not change the original metamodel (just adding new ones), it acts like a

lightweight one, as the extended part can be make available as an independent module and can be easily incorporated in existing KDM tools.

Another important and interesting point here is the following. KDM is not a metamodel intended to serve as base for diagrams, like UML. While UML instances are usually created by humans, KDM instances are system representations created by parsers and processed by tools. So, lightweight profiles make much more sense in the context of UML than in the context of KDM.

B. *Aspect-Oriented Profile*

The main decision before the creation of KDM-AO was to choose an UML profile which was broad enough to represent all the AO concepts. In this sense, we conducted a literature review to identify Aspect-Oriented metamodels and UML profiles that could be considered good candidates. We had analyzed several proposals [10] [11] [13] [14] [15] [16] [17] [18] [19] [31], but the Evermann's profile was considered the most suitable one because it incorporates the level of details that we are interested in [8].

Although this profile has been primarily proposed for AspectJ language, it incorporates all the AO generic concepts. Observing aspect-oriented languages like AspectJ, AspectC++ and AspectS, it is possible to notice that Evermann´s profile involves all of the details presented in these languages, obviously using different terminology. It is like a superset for aspect-oriented programming. This is not a problem because if we want to represent an AspectS program using Evermann´s elements the only possible problem is that some elements will keep empty. However, this is acceptable in the KDM philosophy, since it has an element called *ClassUnit* for representing classes, but we can also create KDM instances for procedural systems. Therefore, this type of element would not be created.

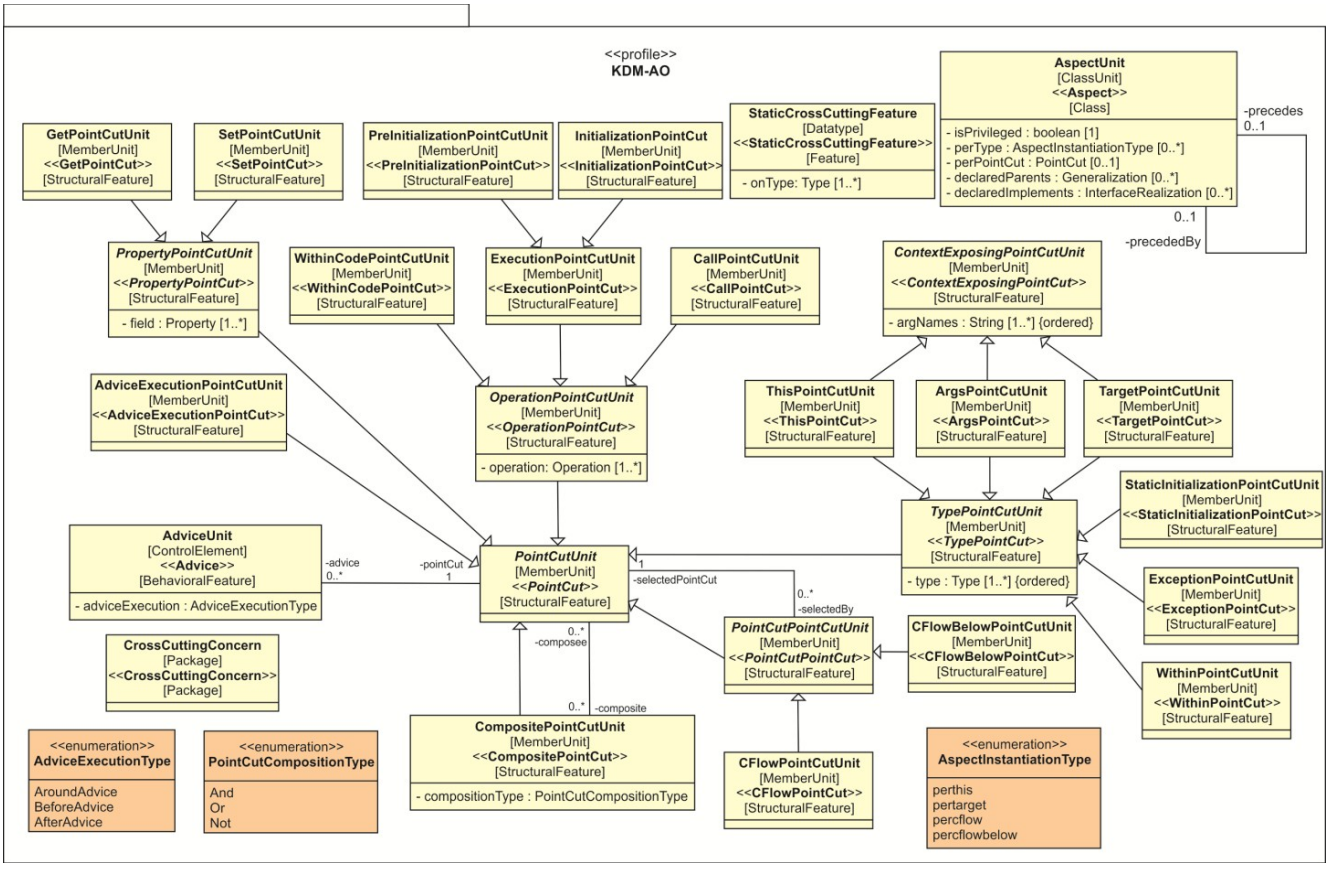

Figure 2. KDM-AO and Evermann's Profile (Adapted).

Fig. 2 shows both the Evermann's profile and the KDM-AO. Each class/element has four words in its first compartment. The first word (in bold) represents the name of the metaclass we have created in our extension, for example, *AspectUnit*. The second word inside the brackets is the KDM superclass we have chosen to make the actual element extends from. For example, we have decided to make our new metaclass *AspectUnit* extends the KDM metaclass *ClassUnit*. Below the mentioned elements, we also have two more words representing the Evermann's profile. For example, the stereotype *<<Aspect>>* created by Evermann extends the UML metaclass Class. These are the third and fourth words. So, in this figure, each element/class represents either a KDM-AO's metaclass, or an Evermann's stereotype, or an *enumeration* of values.

As previously mentioned, the profile proposed by Evermann uses specific AspectJ elements. However, the higher level elements are common to all AO languages, such as Aspect, Advice and Pointcut. Another point that has also guided our decision in choosing Evermann profile was that it had already been reviewed and modified by Gottardi [12], which allowed us to get a better view of its construction.

III. ASPECT-ORIENTED KDM

In this section we present some details of KDM-AO, which can be seen in Fig. 2. The first pair of brackets ([ ]) under the name of the each element exhibits the name of the KDM metaclass that was extended.

One of the biggest challenges when extending metamodels is to know if the metaclasses chosen as base for a new element is the most suitable ones. As Evermann´s profile elements had already previously been mapped to UML metaclasses (extending them through stereotypes), our main task was to identify KDM metaclasses that had similar characteristics to those ones used by Evermann. Due to that, it had been necessary to develop a mapping between both metamodels (UML and KDM), which can be seen in Table 1.

This mapping table identifies KDM metaclasses possessing similar characteristics to UML metaclasses. Some metaclasses can be direct mapped, such as Class from UML, which can be easily mapped to the *ClassUnit* metaclass from KDM. Both present the same goal and characteristics; representing classes in an object-oriented context. However, as KDM aims to represent lower-level details than UML, some UML metaclasses do not have just one candidate in the KDM side. This is the case of Property. This UML metaclass has three possibilities in KDM: *StorableUnit*, *ItemUnit* or *MemberUnit*. *StorableUnit* represents primitive type variables; *ItemUnit* represents registers and *MemberUnit* represents associations with other classes. This abstraction gap occurs because the Code package of KDM is in a lower abstraction level than UML.

TABLE I. KDM-UML MAPPING

UML Element	KDM Element	Differences
Class	*ClassUnit*	The metaclass Class (UML/ Basics package) has four properties: *isAbstract*, *ownedProperty*[*], *ownedOperation*[*] and superClass. The *ClassUnit* element, from *Code Package* encompasses all of these properties through the *AbstractCodeElement* class. A *ClassUnit* may have any attribute whose type is a concrete class of *AbstractCodeElement*, like *StorableUnit*, *MemberUnit*, *ItemUnit*, *MethodUnit*, *CommentUnit*, *KDMRelationships*, etc.
Operation	*MethodUnit*	*Operation* (UML/Basics package) is a behavioral element that has the following properties: class (specifies the owner class), *ownedParameter* (Operation's parameters) and *raisedException* (Operation's exceptions). The *MethodUnit* class is the ideal element to represent Operations because it is a behavioral KDM element capable to represent the most diverse programming languages operations. *MethodUnit* has attributes like *kind* (defines the kind of the operations, for example: abstract, constructor, destructor, virtual, etc.) and *export* (defines the access modifiers, for example: public, private and protected)
Property	*Storable, Member* or *ItemUnit*	*Property* (UML) represents variables in general (local variables, global variables, arrays, associations, etc.), while KDM has an element for each kind of *Property*: primitive type variable (*StorableUnit*), records and arrays (*ItemUnit*) class members (*MemberUnit*)
Package	*Package*	A *Package* on UML (Basics package) is very similarly to a KDM Package (*Code Package*). Both are containers for program elements, like classes, and others code elements. A *Package* could have one or more classes, and a class could have many others elements, like *methods*, *properties*, *comments*, etc.
Structural Feature	*DataElement*	*StructuralFeature* (UML/*Core*::*Abstractions* package) is an abstract metaclass that can be specialized to represent a structural member of a class, like a property. The KDM has the *DataElement* class (*Code package*), that can be specialized to *StorableUnit*, *MemberUnit* or *ItemUnit*.
Behavioral Feature	*Control Element*	*BehavioralFeature* (UML/ *Core::Abstractions* package) is an abstract metaclass that can be specialized to represent behavioral members of a class. The equivalent class on KDM is the *ControlElement*, an abstract class that can be specialized to represent callable elements, including behavioral elements like *MethodUnit*.
Parameter	*Parameter Unit*	*Parameter* (UML/ *Core:Abstractions*) is an abstract metaclass to represent the name and the type of the element that will be passed by parameter in a behavioral element. On the KDM we can use the *ParameterUnit* class. This metaclass can also represent the name, type, position of the parameter in the signature and the kind of paremeter (value or referece)
Relationship	*KDM Relationship*	Both *Relationship* and *KDMRelationship* metaclasses are abstract metaclasses that can be specialized to represent some kind of relationship between two elements, like *Aggregation*, *Generalization*, etc.
...	...	...

In Table 1 it is possible to see the existing relation between the metaclasses and also some comments about it. As KDM is a metamodel much broader than UML, most of the relations just make sense considering the Code Package of KDM, as this package is the one that aims to represent classes, attributes, methods, relationships and other static characteristics. The other KDM packages are more concentrated on details that are absent in UML, like Graphical

User Interface (GUI), architecture and conceptual elements. Because of space limitations, our mapping table shows just the main elements we have used in our KDM-AO extension. However, notice that we mapped all the classes from Evermann's profile.

Based on this mapping, we developed our KDM-AO by creating a new KDM metaclass for every stereotype presented in the Evermann´s profile but exchanging the metaclasses that was extended. For example, if a stereotype in the Evermann´s profile extends the Class metaclass, we then created a new corresponding metaclass in KDM (naming it in a similar way) and make it extends the *ClassUnit* metaclass from KDM.

As can be seen in Fig. 2, the main object-oriented elements (concepts) of Evermann´s profile are represented for higher level classes/stereotypes, which are: CrosscuttingConcern, Aspect, Advice, Pointcut and StaticCrossCuttingFeature. The remainders are subclasses of these higher level elements, representing subtypes. In this section we describe the corresponding elements we have created for each of the main elements. As it was presented earlier, in our KDM-AO, the name of most of our elements ends with the word Unit, for example, *AspectUnit*, *AdviceUnit* and *PointCutUnit*. That is the way we have used to differentiate between our elements from Evermann´s ones.

In Evermann´s profile, the CrosscuttingConcern element extends the Package UML metaclass and aims to represent the existence of a crosscutting concern like persistence, security and concurrency. In our KDM-AO this element extends the *Package* metaclass. This KDM metaclass represents a package in which is possible to encapsulate Aspects, Classes and others elements.

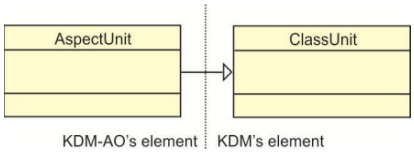

Figure 3. AspectUnit.

AspectUnit is our element for representing aspects, which extends the *ClassUnit* KDM metaclass (Fig. 3). We decided to extend this metaclass because aspects have all the characteristics classes have, besides pointcuts, advices and intertype declarations. From Fig. 3 to Fig. 6, we decided to omit the attributes/properties because all of them can be seen in the corresponding class in Fig. 2.

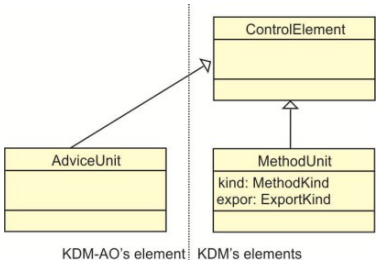

Figure 4. AdviceUnit.

Our element for representing advices is *AdviceUnit* (Fig. 4), which extends the *ControlElement* metaclass. Knowing that advice is an element that specifies behavior, we could consider it like a method. However, advices do not have neither access specifiers (public, private, protected) nor types (constructor, destructor, etc). Because of that we have decided do not make it extends MethodUnit.

PointCutUnit is our element for representing pointcuts. According to Evermann´s profile, pointcut is a structural element and extends the UML metaclass *StructuralFeature*. KDM has also a class for representing structural characteristics called *DataElement*, which is an abstract metaclass. Its descendents are *StorableUnit*, *MemberUnit* and *ItemUnit*. As *StorableUnit* and *ItemUnit* cannot be abstract, *MemberUnit* was chosen to be the superclass of *PointCutUnit*. Besides, another reason for extending *MemberUnit* was that pointcuts can crosscuts other classes and *MemberUnit* is the KDM metaclass used to denote references to other classes. The relations in which these classes are involved can be seen in Fig. 5.

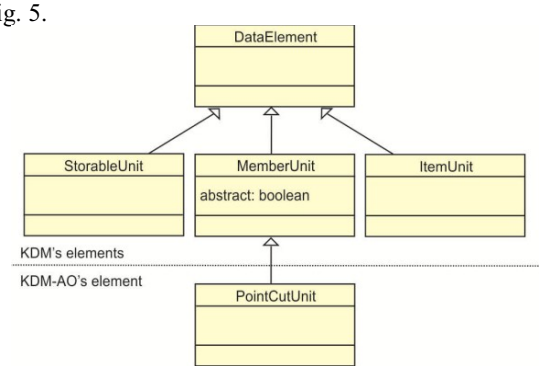

Figure 5. PointCutUnit.

StaticCrossCuttingFeature is our element for representing intertype declarations. In our KDM-AO we have decided to extend two KDM metaclasses: *StorableUnit* e *MethodUnit*. In this way, *StaticCrossCuttingFeature* is able to represent structural and behavioral characteristics. Therefore, an instance of *StaticCrossCuttingFeature* can be an attribute or a method (see Fig. 6).

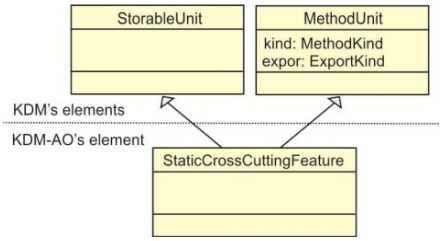

Figure 6. StaticCrossCuttingFeature.

Implementation Details. In order to create the KDM-AO, we have used the Eclipse IDE and Eclipse Modeling Framework (EMF) plug-in, which allows visualizing and editing the KDM metamodel in the Ecore format, available at the OMG website.

Each profile class is represented by means of EMF elements: *Eclass*, *EEnum*, *EPackage*, *EAtribute* and *EReference*. In Fig. 2, almost every class is represented inside

the metamodel for the *EClass* element. The elements denoted as <<*enumeration*>> are represented by the elements *EEnum*. The attributes inside the classes are recreated by the element *EAttribute* and the relationships between the profile classes are specified by the elements *EReference*. Fig. 7 shows one of our *AspectUnit* metaclass represented in the KDM metamodel. It is possible to see in part A the class attributes (*isPrivileged*, *perType*, *perPointCut*, *declaredParents* e *declaredImplements*) and relationships (*precedes* e *precededBy*). Part B shows that metaclass already introduced inside the KDM metamodel along with all of its attributes.

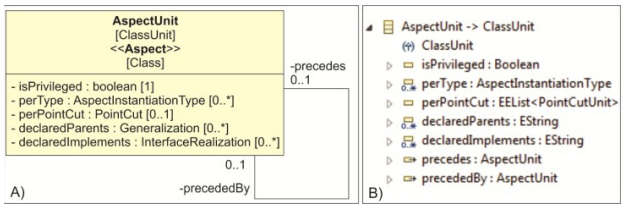

Figure 7. KDM-AO in EMF.

At every new added element there is a set of properties in which some have already default value and other do not, that is, it needs to be fulfilled. For instance, when adding a new *EClass* element, the main properties that must be informed are *Name* and *ESuperTypes* (super classes inherited by the new element). In Fig. 8 we show the properties belonging to *AspectUnit* metaclass. As long as all new metaclasses have been created in KDM, we generated a plug-in called KDM-AO plugin which allows the creation of KDM-AO instances.

Property	Value
Abstract	false
Default Value	
ESuper Types	ClassUnit -> Datatype
Instance Type Name	
Interface	false
Name	AspectUnit

Figure 8. AspectUnit properties.

IV. CASE STUDY

In this section we present a case study showing that the KDM-AO can be used to support a modernization process based on Crosscutting Frameworks (CFs) [3][22]. CFs are aspect-oriented frameworks that encapsulate in a generic way just one crosscutting concern, like persistence, security and cryptography [3][22]. CFs are composed of concrete and abstract aspects and also concrete and abstract classes. Most of them heavily rely on intertype declarations, dynamic crosscutting and well known aspect-oriented idioms like Container Introduction and Marker Interface [27].

The modernization scenario we regard here considers the existence of i) an instance of KDM representing a legacy system (here called “legacy KDM” or "base model") that needs to be modernized; ii) one or more instances of KDM representing CFs available in a repository; and iii) one or more KDM instances representing the elements of instantiation, i.e., concrete classes and aspects created by the application engineer to couple the CFs to a base code or base model (in this case, the legacy KDM).

In this case study, we have modernized a management system of a CD/DVD shop. The modernization goal was to modularize the persistence concern with aspects. As our group has some experience with Crosscutting Frameworks, the idea was to use a Persistence CF previously developed in this process. Doing that, we would be validating the KDM-AO in representing aspects and also in representing CFs.

Note that the focus of this paper is to show that it is possible to represent AO concepts with extended KDM. It is out of our scope mining the legacy KDM looking for crosscutting concerns, remove them or even provide a tool that facilitates the coupling of CFs. Therefore, the first step was to obtain a KDM instance representing the CD/DVD Shop system. This was done using MODISCO [25], which has a parser that automatically transforms Java source code into KDM XMI instances. The second step was concerned in obtaining an instance of our KDM-AO for our Persistence CF. This was done using a plug-in developed in this work, called KDM-AO plug-in, in which classes and aspects were converted into a KDM instance representing the CF.

Because of space limitations, in this section are shown only the main CF´s aspects with the wide variation in the use of elements that were inserted into the KDM metamodel related to the proposed in Evermann´s profile [8].

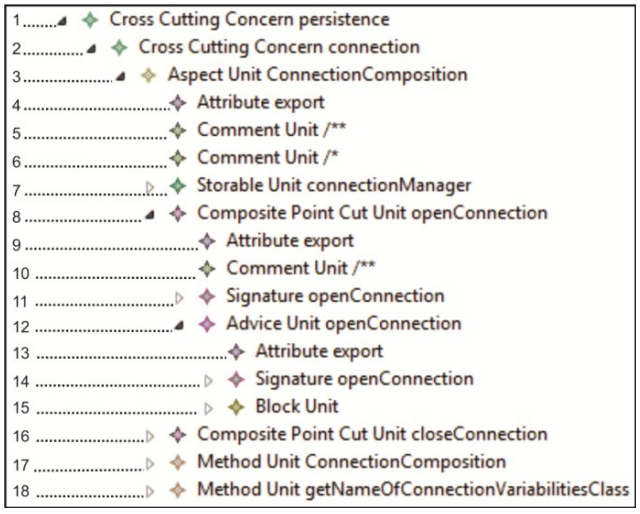

Figure 9. ConnectionComposition.aj in KDM-AO.

In Fig. 9 we can see an abstract aspect of the CF called *ConnectionComposition.aj* which is located inside the package *persistence.connection*. The purpose of this aspect is to provide a base behavior for opening and closing database connections. During instantiation, one needs to provide concrete implementations for the abstract pointcuts `openConnection()` and `closeConnection()`. This aspect has in your body an attribute, two abstract pointcuts, a concrete and one abstract operation and two advices.

The visualization shown in Fig. 9 is possible because of the use of KDM-SDK plug-in that allows one to edit XMI

models in accordance with the KDM metamodel [6]. However, you can also view the file generated by the plug-ins KDM-SDK and KDM-AO-plugin in XMI version.

Each line in Fig. 9 contains the element type and then its value. For example, in the first line we can visualize the existence of a *CrossCuttingConcern* element; whose value is persistence, i.e., this is an instance of the *CrossCuttingConcern* metaclass. Line 3 displays the name of the aspect that is being modeled here; initially the type (*AspectUnit*), then its value (*ConnectionComposition*).

The element *Attribute export* (line 4) is used to store the visibility (Public, Private and Protected), as well as indicate if the element is abstract or concrete. This element is used to represent classes, aspects, methods, pointcuts, advices among others elements that allow this type of statement. The element *StorableUnit* (line 7) is used to declare variables and *PointCutCompositeUnit* (lines 8 and 16) is used to represent concrete or abstract pointcuts of an aspect.

The element *Signature* (lines 11 and 14) receives the same name that the element does and has the function of storing the parameters that were passed in Pointcuts, Methods and Advices. *AdviceUnit* (line 12) represents an advice that was declared in the aspect. It is essential to fill the Advice Execution property because this property declares what kind of advice that element represents (After, Before or Around). In Fig. 9, the element *BlockUnit* (line 15) is the body of advice and you can represent snippets such as `try/catch`, among others. Comment Unit (lines 5, 6 and 10) stores comments that have been made in the source code and *MethodUnit* (lines 17 and 18) allows representing methods in the aspect.

It's Important to say that one source code in AspectJ consists of Java code and aspect code. Elements such as *MethodUnit, CommentUnit*, *Signature*, *BlockUnit* and *Attribute Export* already exist in the KDM metamodel and are being used to make the representation of the common elements of the Java language within the aspect.

```
1  <codeElement xsi:type="code:AspectUnit" name="OORelationalMapping">
2    <attribute tag="export" value="public abstract"/>
3    <attribute tag="export" value="private"/>
4    <ownedElement xsi:type="code:StaticCrossCuttingFeature" name="" ext="">
5      <ownedElement xsi:type="code:StorableUnit" name="tableName" type="/0/@model.0/@codeI
6        <attribute tag="export" value="public"/>
7        <ownedRelation xsi:type="code:HasValue" to="/0/@model.1/@codeElement.42"/>
8      </ownedElement>
9      <onType>PersistentRoot</onType>
10   </ownedElement>
11   <ownedElement xsi:type="code:StaticCrossCuttingFeature" name="">
12     <ownedElement xsi:type="code:MethodUnit" name="getID" type="/0/@model.0/@codeElement
13       <attribute tag="export" value="public abstract"/>
14       <codeElement xsi:type="code:Signature" name="getID">
15         <parameterUnit type="/0/@model.0/@codeElement.2/@codeElement.0" kind="return"/>
16       </codeElement>
17     </ownedElement>
18     <onType>PersistentRoot</onType>
19   </ownedElement>
```

Figure 10. A snippet of the aspect *OORelationalMapping.aj* in XMI format.

In Fig. 10 is shown the *OORelationalMapping* aspect as an instance of KDM-AO in XMI. This aspect aims to introduce (by intertype declaration) dozens of persistence methods in persistent classes of the application. In line 1, there is a declaration of the *OORelationaMapping,* which is an *AspectUnit*. Inside it, there are two Intertype Declarations through *StaticCrossCuttingFeature* element (lines 4 and 11). This kind of statement allows someone to insert properties and operations in other elements, such as interfaces, aspects and classes, just filling in the values in the *onType* attribute (lines 9 and 18). The first *StaticCrossCuttingFeature* (line 4) that appears is inserting a *StorableUnit* (line 5) named *tableName* in *PersistentRoot* interface. The second one (line 11) is inserting a *MethodUnit* element (line 12) named *getID* interface into the same Interface (*PersistentRoot*). In Fig. 11 is the equivalent AspectJ source code represented in the XMI in Fig. 10.

Figs. 9 and 10 showed that it is possible to represent and store KDM instances that represent aspects of CF's or conventional aspects. Another essential activity during the reuse of CFs is to perform the instantiation process and the coupling of the CF to a code base. This is done specializing concrete operations and pointcuts.

```
public abstract aspect OORelationalMapping {
   public String PersistentRoot.tableName = "";
   [...]
   public abstract int PersistentRoot.getID();
```

Figure 11. A snippet of the aspect *OORelationalMapping.aj* source code.

In our case study, it was necessary to create four concrete aspects and one class manually to perform the coupling of the CF to the *CDStore* application. The aspects created were *MyOORelationalMapping*, *MyConnectionCompositionRules*, *MyDirty* and *MyAspect* and the class was *MyConnectionVariabilities*. The *MyConnectionVariabilities* class stores information about the database; the aspect *MyOORelationalMapping* declares classes of the base application that should receive persistence methods; the aspect

MyConnectionCompositionRules specifies the points at the connection to the database will be opened and closed. Finally the aspect *myDirty* and *myAspect* that are abstracts and extend aspects of the CF.

Fig. 12 shows the *MyOORelationalMapping* aspect (lines 1, 2 and 3), whose name can be seen on line 2. Inside this aspect created by the application engineer there are declare parents statements informing application classes that they must extend a CF interface called *PersistentRoot*. This is done so that all classes receive application persistence operations defined in this interface.

```
1  <codeElement xsi:type="code:AspectUnit"
2                  name="MyOORelationalMapping"
3                  isAbstract="false">
4    <attribute tag="export" value="public"/>
5    <ownedRelation xsi:type="code:Imports"
6                     to="CrossCuttingConcern persistence"
7                     from="AspectUnit MyOORelationalMapping"/>
8    <ownedRelation xsi:type="code:Imports"
9                     to="CrossCuttingConcern application"
10                    from="AspectUnit MyOORelationalMapping"/>
11   <ownedRelation xsi:type="code:Extends"
12                    to="AspectUnit OORelationalMapping"
13                    from="AspectUnit MyOORelationalMapping"/>
14   <ownedRelation xsi:type="code:InterfaceRealization"
15                    to="InterfaceUnit PersistentRoot"
16                    from="ClassUnit Music"
17                    Name="Music-PersistentRoot"/>
18   <ownedRelation xsi:type="code:InterfaceRealization"
19                    to="InterfaceUnit PersistentRoot"
20                    from="ClassUnit Purchase"
21                    Name="Purchase-PersistentRoot"/>
22   <declaredParents>Music-PersistentRoot</declaredParents>
23   <declaredParents>Purchase-PersistentRoot</declaredParents>
24 </codeElement>
```

Figure 12. A snippet of the aspect *MyOORelationalMapping*.aj in XMI format.

In lines 5 to 7 and 8 to 10 are shown two *Imports*, the first package is the *persistence* (CF package) and the second is the *application* package (base application package). The lines 11 to 13 represent the *Extends* element that stores the information that the aspect *MyOORelationalMapping* extends the behavior of the *OORelationalMapping* aspect, present in the CF.

In lines 22 and 23 of this aspect can be visualized the use of the *declareParents* element, this element stores the name of a relationship between a base application class and a CF's aspect or interface.

To specify this relationship in a model, it's necessary to use an element that can store the name of the base application class, the name of the CFs aspect or interface and the name of the relationship between them. In KDM, the element capable to do this representation is the *Implements*, however, he does not have a name property, thus it was necessary to extend this element and add the "name" property. This new element created from Implements was called *InterfaceRealization*.

Lines 14 to 17 represent an instance of the element *InterfaceRealization* where line 15 shows the CF *PersistentRoot* interface, line 16 shows the Music basic application class and the name of the relationship between them can be seen in line 17. Another *InterfaceRealization* instance can be seen in lines 18 to 21.

The second snippet in the source code instantiation to be shown are the pointcuts that open and close the connection to the database, present on the aspect *myConnectionCompositionRules*. In Fig. 13 is shown a snippet of a XMI file that contains the element pointcut *openConnection* where is possible to see its main elements, like *CompositePointCutUnit*, *ExecutionPointCutUnit* and *ParameterUnit*. The *CompostitePointCutUnit* element (line 5) is the encapsulation of all pointcuts that represent the *openConnection* (line 6). The *ExecutionPointCut* (Line 9) is the pointcut that crosscutting the *main* method (line 13) from *FindSomeCDs* class (Line 11). Finally, the *ParameterUnit* (lines 16 to 19) store the pointcut parameters.

```
1  <ownedElement xsi:type="code:AspectUnit"
2                  name="myConnectionCompositionRules"
3                  isAbstract="false">
4    <attribute tag="export" value="public"/>
5    <ownedElement xsi:type="code:CompositePointCutUnit"
6                    name="openConnection"
7                    compositeType="OR">
8      <attribute tag="export" value="public"/>
9      <ownedElement xsi:type="code:ExecutionPointCutUnit"
10                     type="//@model.0/@codeElement.1/@codeEleme
11       <codeElement xsi:type="code:ClassUnit" name="FindSomeCDs">
12         <attribute tag="export" value="public"/>
13         <codeElement xsi:type="code:MethodUnit" name="main">
14           <attribute     tag="export" value="public static"/>
15           <codeElement xsi:type="code:Signature" name="main">
16             <ownedElement xsi:type="code:ParameterUnit"
17                           type="//@model.0/@codeElement.1/@c
18                           kind="return"/>
19             <parameterUnit name=".." ext="" kind="unknown"/>
```

Figure 13. A snippet of the aspect *MyConnectionCompositionRules*.aj in XMI format.

With the realization of this case study was possible to ascertain the suitability of the extension developed to represent the most important characteristics and specificities of a CF implemented in AspectJ.

Lower level speficifications. To represent a generic *pointcut* with our extension, it is only needed create an instance of *OperationPointCutUnit* (Fig. 1) and inform the parameters that crosscut the base system. But if a more specific *pointcut* has to be represented, it's possible to create an instance of a more specific *pointcut*. For example, *GetPointCutUnit* and *SetPointCutUnit* are specials kinds of *PointCutUnit* that represent field accesses.

Another example is the control flow of a join point. A control-flow pointcut always specifies another pointcut as its argument. There are two control-flow pointcuts, and in our extension they are represented by *CFlowPointCutUnit* and *CFlowBelowPointCut*. The first pointcut captures all the *OperationPointCutUnit* in the control flow of the specified *PointCutUnit*, including the *OperationPointCutUnit* matching the *PointCutUnit* itself. The second *PointCutUnit* excludes the *OperationPointCutUnit* in the specified *PointCutUnit* [30].

Fig. 14 shows how the mentioned *PointCutUnits* can be represented in the KDM-AO plug-in. Lines 2 and 3 represents the *GetPointCutUnit* and *SetPointCutUnit*, representing that the *accountBalance* field will be crosscut when it is read or write. The *CompositePointCutUnit* (lines 4 and 7)

encapsulates the *PointCutUnits,* allowing the application engineer specify the points of the base system that will be affected by *PointCutUnits*. Line 6 shows a *CallPointCutUnit* that is modified by a *CFlowPointCutUnit* (line 5) and Line 9 shows a *ExecutionPointCutUnit* that is modified by a *CFlowBelowPointCutUnit* (line 8).

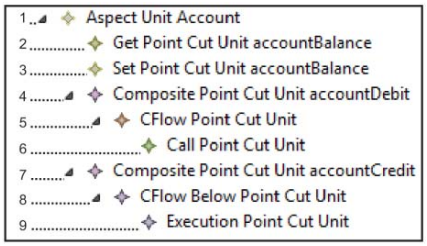

Figure 14. Lower level specifications example in KDM-AO plug-in.

There are others possible representations of pointcuts supported by our extension, and the level of details of a KDM-AO instance will depend mainly on the application engineer and the parser that creates the instance.

V. Related Works

The research work most related to ours is the KDM AO extension presented by Mirshams [9]. As we have done here, this author also created a heavyweight KDM extension for aspect-oriented programming. There are three main differences between our works. Firstly, while Mirshams has based her extension in an aspect model created by herself, we have created our extension based on a very well known profile for aspect-oriented programming. Evermann´s profile encompasses all the AO concepts presented in AspectJ and in other less known aspect-oriented languages, like Aspect C++ and AspectS [8].

The second difference is the level of abstraction of our extensions. The aspect model used by Mirshams contains much less elements than Evermann´s profile. That means our extension is able to represent both a high level (using the most generic metaclasses) and a low level (using most specific metaclasses) view of the system. In her case, just a higher level view is possible. The third difference is that her work is limited to dynamic crosscutting as there are no elements for representing intertype declarations. However, despite all of these differences, the main similarity is that we have used the same KDM metaclasses she has used too.

Another KDM extension is presented by Baresi and Miraz [28]. They proposed a heavyweight KDM extension to support Component-Oriented MOdernization (COMO). COMO is a metamodel that supports traditional concepts of software architecture, allowing to attach software components in KDM. Using their extension it is possible to replace or add parts of a system. Unlike we have done here, in their paper they had not used an existing profile as the starting point for creating their extension - they combined another metamodel to the KDM.

COMO extends some high level metaclasses of KDM, such as *KDMModel*, *KDMEntity* and *KDMRelationship*. These classes are the base of their extension and provide the link between KDM and COMO metamodels.

The main similarity with our work is that they have also performed a heavyweight extension in KDM. As a main difference, the extension presented by them only extended high level elements of KDM, while in our solution we have use more specific elements such as *ClassUnit* and *MemberUnit*.

VI. Discussions and Conclusions

As we have commented in Section II, a heavyweight extension can change the original metamodel or simply add new metaclasses. We have opted for this second choice because it facilitates the reuse of our extension in other contexts. Besides, only by using heavyweight extensions is possible to guarantee some level of correctness in model-level. Otherwise this responsibility must be transferred to tools.

By means of our case study, it is fairly evident that our extension can represent all AOP elements. However, as we have not carried out a complete case study to gauge how reliable our extension is to represent aspects concepts in other programming languages, such as AspectC++, we consider this is a limitation of our extension. Nevertheless, to mitigate this limitation, the elements of AspectC++ and AspectS were analyzed. Consequently, we conclude that there are enough elements in our extension that can be used to represent source code in both AspectC++ and AspectS.

Apart from Mirsham´s work [9] we did not find another work that has extended KDM for aspect-oriented programming. Her extension is also a heavyweight solution and does not include inter-type declarations. Besides, her solution does not allow representing lower level concepts.

Another contribution here is to show a preliminary mapping between UML and KDM which can be used to turn UML profiles into to KDM extensions. In our case, we turned an AO UML profile into a KDM heavyweight AO extension. However, considering the mapping shown in Table 1, any UML profile could be transformed. This is quite useful because in Model-Driven Environments, systems that are represented as KDM instances will need to be visualized as class diagrams.

Although our case study has shown just the ability of KDM-AO to represent existing code (reverse engineering), it can also be used in forward engineering for code generation. In this case, it seems to be more appropriate than the Mirsham's profile, since ours includes lower level details.

When conducting our case study using CFs, we have noticed that our extension would be more useful and more expressive if it had also metaelements for representing CF characteristics, like hot spots, frozen spots and other framework characteristics. We intend to perform these modifications in a future work [3].

Another interesting work we intend to conduct in the future is to compare our heavyweight extension with a lightweight one. Currently, we are already developing a lightweight version of our extension. However, we can already anticipate that one of the biggest drawbacks of the lightweight version is the possibility of including erroneous relationships in the model. We are also planning to carry out an experiment to list the vantages and disadvantages of each extension.

As other future works, we aim to conduct others case studies using AspectC++ and AspectS in order to test the

KDM-AO extension, with the objective of evaluating the issue of platform independence. Another future work that can be done is to check if there is some other element to be added to the profile, taking into account new additions to the aspect-oriented programming from 2007 to the current year.

By conducting this research we have noticed that the power of model-driven modernization is greatly influenced by the capacity of representing specific concepts in a proper and suitable way.

As we have shown, the absence of aspectual concepts in the original KDM prevent the realization of aspect-oriented modernizations, or at least, makes it very hard. The same occurs when we consider other fields/domains, such as: web services, embedded systems, business processes, fault tolerance, testing, etc. All of these subareas has already UML profiles, available at OMG [26] [27], aiming to represent specific details/concepts/abstractions in a more precise way. Therefore, our mapping table can be easily employed to create KDM extensions from all of these UML profiles, as we have exemplified here.

ACKNOWLEDGEMENTS

Bruno M. Santos and Raphael R. Honda would like to thank CNPq for sponsoring our research. Rafael S. Durelli and Valter V. de Camargo would like to thank the financial support provided by FAPESP, processes numbers 2012/05168-4 and 2014/14080-9, respectively.

REFERENCES

[1] G. Visaggio, "Ageing of a data-intensive legacy system: symptoms and remedies," Journal of Software Maintenance 13. 2001, pp. 281–308, doi: 10.1002/smr.234.

[2] Architecture-Driven Modernization, 2014.
Document omg/ http://adm.omg.org/.

[3] V. V. Camargo and P. C. Masiero, "Frameworks Orientados a Aspectos," XIX Simpósio Brasileiro de Engenharia de Software, Uberlândia. 2005, pp. 200-216.

[4] D. L. Parnas, "Software aging," ICSE '94 Proc. of the 16th international conference on Software engineering, Los Alamitos, CA, USA, 1994, pp. 279-287.

[5] K. Normantas, S. Sosunovas and O. Vasilecas, "An Overview of the Knowledge Discovery Meta-Model," Proc. of the 13th International Conference on Computer Systems and Technologies - CompSysTech'12, 2012, pp. 52-57.

[6] Knowledge Discovery Meta-Model. KDM Guide, August 2011.
Document omg/formal/2011-08-04.

[7] V. V. Camargo, P. C. Masiero, "An Approach to Design Crosscutting Framework Families," ACP4IS 08, Brussels, Belgium, 2008.

[8] J. Evermann, "An overview and an empirical evaluation of UML: an UML profile for aspect-oriented frameworks," Workshop AOM '07, Vancouver, British Columbia, Canada, 2007.

[9] P. S. Mirshams, "Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming," 79 p. Dissertation (Master in Applied Science in Software Engineering) – Computer Science Department and Software Engineering, University of Montreal, Quebec, Canada, 2011, unpublished.

[10] M. Kande, J. Kienzle, and A. Strohmeier, "From AOP to UML - a bottom-up approach," Proc. of the AOM with UML workshop at AOSD, 2002, 2002.

[11] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli, "A UML notation for aspect-oriented software design," Proc. of the AOM with UML workshop at AOSD, 2002, 2002.

[12] T. Gottardi, R. A. D. Penteado and V. V. de Camargo, "A Process for Aspect-Oriented Platform-Specific Profile Checking," Proc. of the 2011 International Workshop on Early Aspects. New York, NY , USA. 2011.

[13] D. Stein, S. Hanenberg, and R. Unland, "Designing aspect-oriented crosscutting in UML," Proc. of the AOM with UML workshop at AOSD, 2002.

[14] M. Basch and A. Sanchez, "Incorporating aspects into the UML," Proc. of the AOM workshop at AOSD, 2003.

[15] L. Fuentes and P. Sanchez, "Elaborating UML 2.0 profiles for AO design," Proc. of the AOM workshop at AOSD, 2006.

[16] E. Barra, G. Genova, and J. Llorens, "An approach to aspect modelling with UML 2.0," Proc. of the AOM workshop at AOSD, 2004.

[17] J. Grundy and R. Patel, "Developing software components with the UML, Enterprise Java Beans and aspects," Proc. of ASWEC 2001, Canberra, Australia, 2001.

[18] C. Chavez and C. Lucena, "A metamodel for aspect-oriented modeling," Proc. of the AOM with UML workshop at AOSD, 2002.

[19] H. Yan, G. Kniesel, and A. Cremers, "A meta model and modeling notation for AspectJ," Proc. of the AOM workshop at AOSD, 2004.

[20] A. Rashid and R. Chitchyan, "Persistence as an Aspect," 2nd International Conference on Aspect Oriented Software Development (AOSD), Boston–USA, March, 2003.

[21] C. F. M. Couto, M. T. O. Valente and R. da S. Bigonha, "Um Arcabouço Orientado por Aspectos para Implementação Automatizada de Persistência," 2º. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'05), evento satélite do XIX SBES, Uberlândia, MG, Brasil, outubro, 2005.

[22] T. Gottardi, R. S. Durelli, O. P. López and V. V. Camargo, "Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort," Journal of Software Engineering Research and Development, 2013, pp. 1-34, doi:10.1186/2195-1721-1-4.

[23] S. Soares, E. Laureano and P. Borba, "Implementing Distribution and Persistence Aspects with AspectJ," 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), November, 2002, pp 174-190.

[24] A. Rausch, B. Rumpe and L. Hoogendoorn, "Aspect-Oriented Framework Modeling," 4th AOSD Modeling with UML Workshop (UML Conference 2003) October, 2003.

[25] H. Bruneliere, J. Cabot, F. Jouault and F. Madiot, "MoDisco: A generic and extensible framework for model driven reverse engineering," IEEE/ACM international conference on Automated software engineering, ACM New York, NY, USA, 2010, pp. 173-174.

[26] Object Management Group. OMG Specifications, April 2014.
Documents omg/ http://www.omg.org/spec/.

[27] S. Hanenberg, "Multi-Design Application Frameworks," Generative and Component-Based Software Engeneering Young Reaearchers Workshop, Erfurt, October 10, 2000.

[28] L. Baresi and M. Miraz, "A Component-oriented Metamodel for the Modernization of Software Applications," 16th IEEE International Conference on Engineering of Complex Computer Systems. 2011.

[29] G. Kiczales *et al.*, "Aspect Oriented Programming," Proc. of 11 ECOOP. pp. 220-242, 1997.

[30] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications, Greenwich (74° w. long.), 2003, pp.75-77.

[31] J. U. Júnior, R. D. Penteado and V. V. Camargo, "An overview and an empirical evaluation of UML-AOF: an UML profile for aspect-oriented frameworks," Proc. of the 2010 ACM Symposium on Applied Computing. 2010, pp 2289-2296.

[32] R. S. Durelli *et al.*, "A Mapping Study on Architechture-Driven Modernization," 15th IEEE International Conference on Information Reuse and Integration, 2014, pp 1-8.