A Combined Approach for Concern Identification in KDM models

Daniel S. M. Santibáñez*, Rafael Serapilha Durelli†, Bruno Marinho* and Valter Vieira de Camargo*
*Departamento de Computação, Universidade Federal de São Carlos,
Caixa Postal 676 – 13.565-905, São Carlos – SP – Brazil
Email: {daniel.santibanez,bruno.santos,valter}@dc.ufscar.br
†Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: rsdurelli@icmc.usp.br

***Abstract*—The several maintenance tasks a system is submitted during its life usually cause its architecture deviates from the original conceived design, therefore software engineers need techniques for recovering the knowledge embedded in legacy systems in order to get a better software comprehension. With the advent of ADM (Architecture-driven modernization), an OMG standard for modernizing legacy software systems, the reverse engineering process follows a model-driven approach using the KDM metamodel (Knowledge Discovery Metamodel) as the cornerstone of the standard. Nevertheless, although ADM provides the process to modernize legacy systems, it does not support the identification and modularization of crosscutting concerns which is a sort of modernization. In order to overcome this disadvantage it is necessary to extend ADM with new techniques and tools. This paper proposes an approach called CCKDM for identifying crosscutting concerns by means a combination of a concern library and a K-means clustering algorithm. The input of the approach is a KDM model instance and the result is the same KDM model with annotated concerns. To provide some evidence of the precision and recall of our approach we conducted an empirical evaluation that involve two well-known systems. In this evaluation we compared our approach with other ones (XScan and Timna) by using three *levenshtein*. The results that we achieved seen to be similar or equal as compared to those approaches.**

***Keywords*—ADM, KDM, crosscutting concerns, concern mining.**

I. Introduction

Software systems are considered legacy when their maintenance costs are raised to undesirable levels but they are still valuable for organizations. However, they can not be discarded because they incorporate a lot of embodied knowledge due to years of maintenance and this constitutes a significant corporate asset. As these systems still provide significant business value, they must then be modernized/re-engineered so that their maintenance costs can be manageable and they can keep on assisting in the regular daily activities.

Several approaches have been developed to support software engineers in the comprehension of systems where reverse engineering (RE) is one of them [1]. RE supports program comprehension by using techniques that explore the source code in order to find relevant information related to functional and non-functional features [2]. In a parallel research line, researchers have been shifted from the typical RE approach to the so-called Architecture-Driven Modernization (ADM) [3]. ADM has been proposed by OMG (Object Management Group) and advocates conducting RE following the principles of MDA (Model-Driven Architecture) [4], i.e., it treats all the software artifacts involved in the legacy system as models and can establish transformations among them. For instance, firstly a reverse engineering is performed starting from the source code and a model instance (**P**lataform **S**pecific **M**odel - PSM) is created. Next successive refinements are applied to this model up to reach a good abstraction level (PSM) in model called KDM (**K**nowledge **D**iscovery **M**etamodel). Upon this model, several refactorings, optimizations and modifications can be performed to solve problems found in the legacy system. Secondly, a forward engineering is carried out and the source code of the modernized target system is generated again.

According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a multipurpose standard metamodel that represents all aspects of the existing information technology architectures. The idea behind the standard KDM is that the community starts to create parsers from different languages to KDM, thus, everything that takes KDM as input can be considered platform and language-independent. For example, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages. The KDM is divided into four layers representing both physical and logical software assets of information systems at several abstraction levels. Each layer is further organized into packages. Each package defines a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to existing legacy systems. However, in this paper we are only interested in the Program Elements layer, which is used to represent a language-independent intermediate representation for programming languages.

On the other hand mining of crosscutting concerns or Aspect Mining as is also known is another important research field that has be exploited in the last years and it is totally related to reverse engineering [5] [6]. The main purpose is to be able to automatically locate existing crosscutting concerns in the source code of a system [7]. They are indispensable for modernization processes because most of the legacy systems suffer from the existence of a lot of crosscutting concerns spread over their architecture.

Although ADM/KDM had been created to support

modernization of legacy systems, to the best of our knowledge there is not research that address the mining of crosscutting concerns using the KDM metamodel. We claim that this is indispensable because once the legacy systems are instantiated by the KDM they tend to: (*i*) have complex architectures with several clones spread out throughout the KDM model, (*ii*) involve several kinds of crosscutting concerns, e.g., patterns, architectural styles, business rules and non-functional properties and (*iii*) be very large, making the manual mining impractical. Therefore, identifying those concerns which are scattered and tangled with others concerns, it could add an additional value to the KDM model, and also for helping engineers in the software comprehension to take better decisions in software maintenance activities. Besides, we also argue that creating a mining approach which takes as input the KDM makes this language-independent, considering the existence of parsers that generate KDM instances from several languages [8] [9] [3].

In order to overcome this limitation, in this paper we present a mining approach for crosscutting concerns based on a concern library and string clustering algorithm which uses the standard metamodel KDM (CCKDM). In other words, the input of our technique is a KDM model and the output is the same model but with annotated concerns. Thus, the software engineers may perform refactorings over the KDM model, modularizing the concerns without touching the source code. Furthermore, to evaluate our technique, we applied the approach CCKDM in three well known systems - HealthWatcher v10, PetStore v1.3.2 and PrograduWeb.The aim of the evaluation is to identify the precision and the recall during the mining of crosscutting concern of "Persistence".

II. Background

Knowledge Discovery Metamodel (KDM) is the key within set of standards [10]. KDM allows standardized representation of knowledge extracted from legacy systems by means of reverse engineering. KDM provides a common repository structure that makes possible the exchange of information about existing software assets in legacy systems. This information is currently represented and stored independently by heterogeneous tools focused on different software assets [4, p. 32]. Figure 1 shows each of the varying views of the existing IT architecture represented by the KDM. For example, the build view, depicts system artifacts from a source, executable, and library viewpoint. Other perspectives include design, conceptual, data, and scenario views.

The Level 0 (L0) encompasses the Infrastructure and Program Elements Layer. Infrastructure Layer consists of the Core, kdm, and Source packages which provide a small common core for all other packages. Program Elements Layer consists of the Code and Action packages providing programming elements such as data types, data items, classes, procedures, macros, prototypes, templates and captures the low level behavior elements of applications, including detailed control and data flow between statements. The Level 1 (L1) cover the Resource Layer which represents the operational environment of the existing software system. For example, the knowledge related to events and state-transition, the knowledge related to the user interfaces of the existing software system and the knowledge related to persistent data, such as indexed files, relational databases, and other kinds of data storage. The

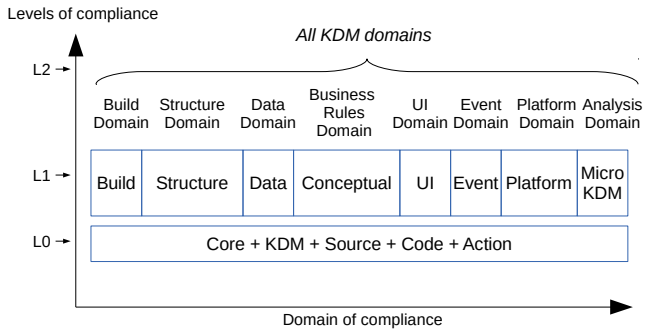

Figure 1: KDM domains of artifact representation (Adapted from Ulrich and Newcomb [4])

Level 2 (L2) cover the Abstraction Layer which represents domain and application abstractions.

As we stated earlier, herein we are only interested in the Program Element Layer - more specifically in the Code Package, which represents the code elements of a program and their associations. Therefore, it is important to dig a little deeper in this metamodel because it is mainly used by our approach in order to identify concerns.

In a given KDM instance, each instance of the code meta-model element represents some programming language construct, determined by the programming language of the existing software system. Each instance of a code meta-model element corresponds to a certain region of the source code in one of the artifacts of the existing software system. In addition, the Code package consists of 24 classes and contains all the abstract elements for modeling the static structure of the source code. However, we are particularly interested in *MethodUnit* and *StorableUnit* packages because they implement crosscutting concerns. In Figure 2 is depicted a chunk of the Code package. It worth to notice that the more important metaclasses used herein are highlighted.

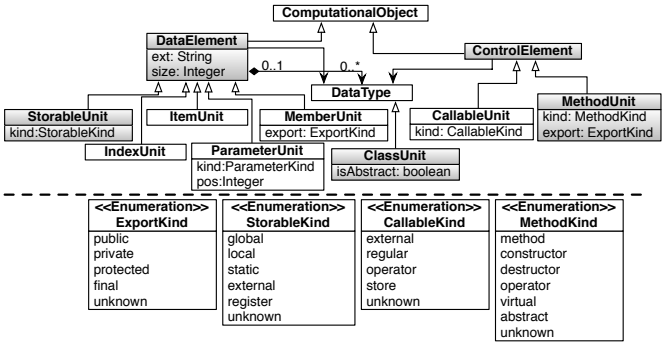

Figure 2: Chunk of the Code Package (OMG Group [11])

As can be seen in Figure 2 the root metaclass is *ComputationalObject* which has two sub-metaclasses, i.e., *DataElement* and *ControlElement*. The former sub-metaclass, *DataElement*, is a generic modeling element that defines the common properties of several concrete classes that represent the named data items of existing software systems, for example, global and lo-

cal variables, record files, and formal parameters. *DataElement* has five sub-metaclasses - *StorableUnit*, *IndexUnit*, *ItemUnit*, *ParameterUnit* and *MemberUnit*. *StorableUnit* is a concrete sub-metaclass of the *StorableElement* meta-class that represents variables of the existing software system. *IndexUnit* class is a concrete subclass of the *DataElement* class that represents an index of an array datatype. Instances of *ItemUnit* class are endpoints of KDM data relations which describes access to complex datatypes. *ParameterUnit* class is a concrete subclass of the *DataElement* class that represents a formal parameter; for example, a formal parameter of a procedure. *MemberUnit* class is a concrete subclass of the *DataElement* class that represents a member of a class type. Finally, the latter, *ControlElement* is a sub-metaclass that contains two sub-metaclasses - *MethodUnit* and *CallableUnit*. *MethodUnit* element represents member functions owned by a *ClassUnit*, including user-defined operators, constructors and destructors. The *CallableUnit* represents a basic stand-alone element that can be called, such as a procedure or a function. As can be seen below the dashed line in Figure 2 there are also the following enumerations: "*ExportKind*", "*StorableKind*", "*CallableKind*", "*MethodKind*", which are sets os literals used as properties of the metaclasses.

Our approach uses those meta-classes to identify the crosscutting concerns rather than using source code. More information about the approach can be seen in Section III.

III. Concern Identification

The developed technique called CCKDM aims to identify code structures into KDM models which may implement crosscutting concerns. The output is an annotated KDM with concerns. Our technique could be classified as a token-based approach, that means analysis of sequences of characters [7].

The Figure 3 depicts the overall process, which is divided into four sub-processes denoted by its corresponding letters at the left side.

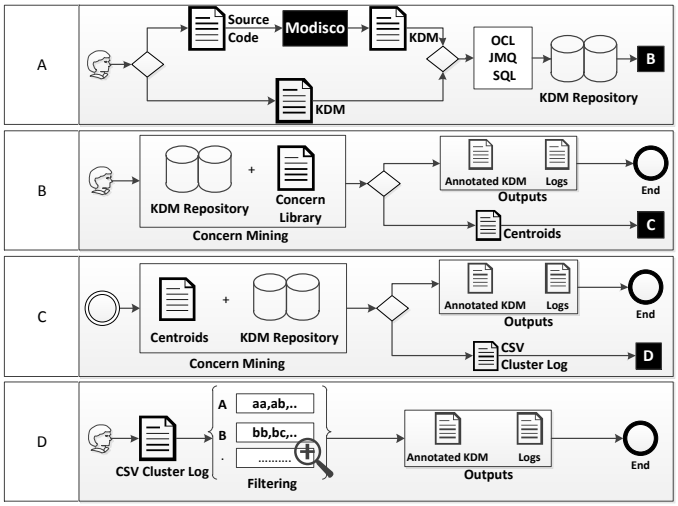

Figure 3: The CCKDM mining process

A. Recovery of Code Structures

The subprocess [A] of the technique starts with user intervention which provide the source code or a KDM file as input to the technique. If the source code is the input, then it is converted to a KDM model by means *Modisco*™ [1] which makes available some APIs called discoverers to convert the source code of a system into a KDM model. After that, the KDM model is queried to recover code structures of the application under study such as methods and properties because they are the most suitable items to find concern seeds [12]. To do that, a combination of three technologies are used to achieve this purpose; a set of simple OCL (Object Constraint Language) queries to recuperate all packages, all classes, all methods and all properties of the model. Then, using the Java Model Query (JMQ) of *Modisco*™ programmatically are retrieved method calls, methods containers, container classes, method signatures, method types and property types. The last step is to persist these elements (strings) into a KDM repository which is a relational database. Figure 4 shows the EER (Enhanced Entity Relationship) model which is composed by eight tables. Table *Class* persists class names. Table *Method* persists method names. Table *Import* persists application imports. Table *Package* persists application packages. Table *GlobalProperty* persists class properties. Table *MethodProperty* persists variables belonging to a certain method. Table *Class_has_Import* persists the relation import/class and table *Method_has_Method* persists method calls.

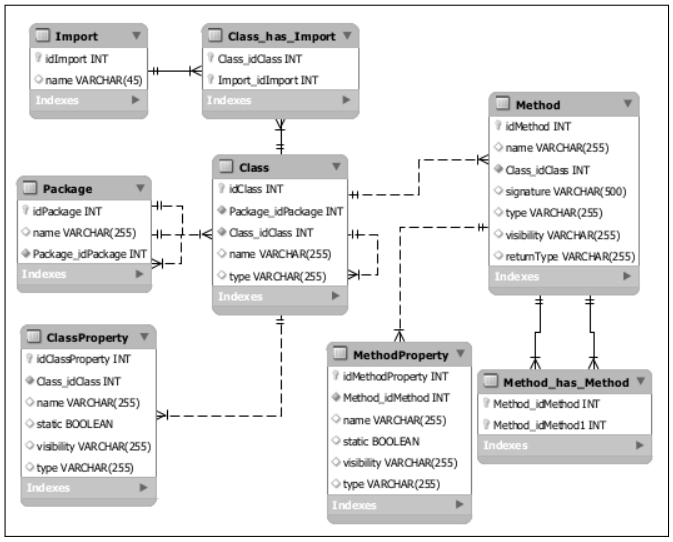

Figure 4: EER model

B. Concern Identification by Concern Library

The subprocess [B] is triggered by user. SQL queries are performed over the KDM repository with information of our concern library which is an XML file containing concern definitions based on [13]. A concern definition is an XML entry composed by a concern name and one or more well-known names of classes implementing the concern, as shown

[1] Modisco provides an extensible framework to develop model-driven tools to support use-cases of existing software modernization, http://www.eclipse.org/MoDisco/.

in Figure 5. Specifically, we are interested in identify possible APIs used by the system because they could be used to implement several concerns. The result is a set of elements (strings) related with some of the classes of the concern library. These strings will be the initial seeds of our approach and the initial information (centroids) for our string clustering approach detailed in the next section. If user does not want to perform the subprocess [C] then the mining process stop and the outputs are log files and the annotated KDM with the identified concern names.

```
▼<ConcernLibrary>
  ▼<Concern name="Persistence">
    ▶<Package name="java.sql">...</Package>
    </Concern>
  ▶<Concern name="Logging">...</Concern>
  ▼<Concern name="Authentication">
    ▼<Package name="javax.security.auth">
       <Element>Destroyable</Element>
       <Element>Refreshable</Element>
       <Element>AuthPermission</Element>
       <Element>Policy</Element>
       <Element>PrivateCredentialPermission</Element>
       <Element>Subject</Element>
       <Element>SubjectDomainCombiner</Element>
       <Element>DestroyFailedException</Element>
       <Element>RefreshFailedException</Element>
     </Package>
    ▼<Package name="java.security">
       <Element>Principal</Element>
     </Package>
    ▼<Package name="javax.ejb">
       <Element>SessionBean</Element>
       <Element>SessionContext</Element>
     </Package>
   </Concern>
 </ConcernLibrary>
```

Figure 5: Concern Library

C. Concern Identification by Clustering

The subprocess [C] performs a string clustering by means a K-means algorithm [14]. The main idea behind this is that "programmers tend to use similar variable names to implement logic programming that meets the same objectives where the contexts are different". In that sense, this subprocess complements the previous one because its objective is to identify names code structures which were not previously identified but have similar names with the strings identified in subprocess B. Thus, we could suppose they implement same concerns. For example, when we applied our approach to HealthWatcher, it identified variables declared by *PersistenceMechanismException* and *PersistenceMechanism* which are not part of Java API.

The strings identified in subprocess [B], which already belong to a certain concern, are the K centroids for our cluster algorithm (*i.e.* the strings where others strings will be clustered). Then, these centroids are compared with method names and property names of the KDM repository by using the *Levenshtein* distance which is a string metric for measuring the difference between two sequences [15]. If the *Levenshtein* resulting value is closer to 1.0 then the compared strings are more similar, on the other hand if it is more closer to 0.0 then the compared strings are more dissimilar. Users can determine the *Levenshtein* distance threshold if they want a more flexible or more restrictive string clustering. Finally, the new identified strings are annotated with their respective concern into the KDM model. The outputs are log files and the annotated KDM.

D. Manual Filtering

The subprocess [D] is optional and gives the possibility to users performing manual filtering of elements identified by the subprocess [C] by means a CSV file generated in the previous subprocess containing all the strings identified by our cluster algorithm with their respective concern name. Users must tag with a "X" at the end of the line in the CSV file if it still be annotated into the KDM file. For all the strings without the "X" tag in the CSV file, the annotation concern into the KDM file will be remove. The outputs are the updated KDM file and log files.

E. Annotating the concerns into the KDM model

The technique performs the annotation activity by introducing a new attribute to the KDM meta-model called *concern*. Thus, methods and properties are annotated with their respective concerns in the following form: *concern="CONCERN"*. Figure 6 shows an annotated example of part of a KDM file for Persistence and Logging. Of course, the new attribute added to the model does not belong to the KDM meta-model so we extended it in order to work with this new feature.

```
    </codeElement>
    <codeElement concern="Persistence" xsi:type="code:
      <attribute tag="export" value="private"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.9"
      </source>
      <comment text="/** &#x0A; */&#x0A;"/>
    </codeElement>
    <codeElement xsi:type="code:StorableUnit" name="me
      <attribute tag="export" value="private"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.9"
      </source>
    </codeElement>
    <codeElement concern="Persistence" xsi:type="code:
model.0/@codeElement.0/@codeElement.0/@codeElement.2/@
      <attribute tag="export" value="public"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.9"
      </source>
    </codeElement>
</codeElement>
<codeElement xsi:type="code:Package" name="model">
  <codeElement xsi:type="code:ClassUnit" name="User" i
    <attribute tag="export" value="public"/>
    <source language="java">
      <region file="/0/@model.2/@inventoryElement.10"
    </source>
    </codeElement>
    <codeElement xsi:type="code:StorableUnit" name="id
      <attribute tag="export" value="private"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.10
      </source>
    </codeElement>
    <codeElement concern="Logging" xsi:type="code:Stor
```

Figure 6: Annotated concerns into KDM

However, we argue this is neither a problem nor a limitation of our approach once according to the KDM's specification it is quite easy to extend it by using the light-weight extension mechanism. The KDM light-weight extension mechanism is a

standard way of adding new "virtual" meta-model elements to KDM [11].

IV. Experimental Study

The goal of our experimental study was to evaluate the effectiveness of our combined technique. Unlike other empirical case studies of concern mining techniques, which describe in detail the kinds of candidates discovered by their tools, we focus here on the precision and recall. We compared empirically CCKDM using 3 *levenshtein* values with *XScan* [16] and *Timna* [17]. The *levenshtein* values were chosen after an statistical analysis which was not included int this work due space limitation reasons.

XScan identifies code units named concern peers and they are detected based on their similar interactions, i.e., similar calling relations in similar contexts, either internally or externally. *Timna* is a framework in which new and existing mining analyses can be easily added and included in the decision as to whether a code segment is a seed of a scattered concern. The main difference is that our technique use a KDM model as input instead of the source code.

A. Subject Programs

In Table II we summarize the applications software under study. All the listed software were used to perform a comparative analysis in terms of precision and recall with others concern mining tools. As we can see, they have a reasonable size (KLOC $\leq$ 9K) which make them suitable to perform a manual analysis of concerns to calculate the recall metric.

Table II: Applications software under study

	System	LOC	Classes	Methods	Properties
1	HealthWatcher v10	8K	118	894	1290
2	PetStore v1.3.2	9K	228	1917	3002

The HealthWatcher is a real health complaint system developed to improve the quality of the services provided by health care institutions. The system has a web-based user interface for registering complaints and performing several other associated operations. PetStore is an application that allows customers to purchase goods via a browser.

B. Empirical Evaluation

In Table I we show the comparison of our technique with *XScan* and *Timna* in terms of precision and recall for Persistence. *XScan* just presents the recall value for HealthWatcher and *Timna* just presents the precision value for PetStore.

Our technique with a *levenshtein* value of 0.3 obtained 100% of precision and recall when we analyzed HealthWatcher. Which means, there are no false negatives and false positives. For *levenshtein* values of 0.3 and 0.4 precision value still remains but the recall decrease. That is normal because higher values of *levenshtein* implies that strings must match more precisely and as a consequence generating false negatives. The scenario changes for PetStore where while the *levenshtein* value increase, the precision increase and the recall decrease. That is because on one hand false negatives increase and on the other hand false positives decrease.

The recall is sensitive to false negatives and on the other hand, precision is sensitive to false positives. If the levenshtein value tends to be high then false positives could increase and if the *levenshtein* value tends to be low then false negatives could increase.

If we compare precision and recall for CCKDM using the three *levenshtein* values with *XScan* and *Timna*, we can see it reaches similar or equal values than the other techniques, so it is possible to say CCKDM has the same effectiveness.

C. Threats to Validity

The lack of representativeness of the subject programs may pose a threat to external validity. We argue that this is a problem that all software engineering research, since we have theory to tell us how to form a representative sample of software. Apart from not being of industrial significance, another potential threat to the external validity is that the investigated programs do not differ considerably in size and complexity. To partially ameliorate that potential threat, the subjects were chosen to cover a broad class of applications. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings. A threat to construct validity stems from possible faults in the implementations of the techniques. With regard to our mining techniques, we mitigated this threat by running a carefully designed test set against several small example programs. Similarly, XSCan and Timna have been extensively used within academic circles, so we conjecture that this threat can be ruled out.

V. Related Work

Concern mining or aspect recommendation has been a popular research topic in recent years. Static mining and history-based mining are two major techniques based on source code analysis. The static technique analyzes source code of a version of software to extract seeds of concerns. A Fan-in value, which is the number of unique callers of each method/function, was first introduced by Marin and others [18] and further generalized by Zhang and others [19] to propose Clustering-Based Fan-in Analysis (CBFA). The history-based mining technique was first adopted by Breu and others [20], who proposed History-based Aspect Mining (HAM). HAM clusters methods/functions that add or remove a call to the same method/function, and groups together methods/functions that are called by the same cluster as concern seeds.

Lengyel et al. [21] proposes a semi-automatic approach to identify crosscutting constraints. The approach uses several algorithms to support the detection of the crosscutting constraints in metamodel-based model transformations. The input of the approach is a transformation (transformation rules and a control flow model), and the expected output is the list of the crosscutting constraints separated as aspects. Van Gorp et al. [22] 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.

Table I: Comparison values of precision and recall for persistence

	CCKDM-0.3		CCKDM-0.4		CCKDM-0.5		XScan		Timna	
Systems	P	C	P	C	P	C	P	C	P	C
HealthWatcher v10	100%	100%	100%	80%	100%	76,11%	-	100%	-	-
PetStore v1.3.2	95%	100%	95,73%	84,15%	98,79%	75,44%	-	-	93.80%	-

The differential of our approach described herein in relation to the others is that our approach mines crosscutting concerns by using KDM instead of another models or source code. It is important to note that to the best of our knowledge there is no previous research that addresses mining crosscutting concerns by using KDM model as input.

VI. Conclusions

We presented a new mining approach for crosscutting concerns called CCKDM which aims to identify and tag crosscutting concerns into KDM models. It is important to note this is the first work in concern mining area that use a standardized model in the context of ADM to perform search of concerns and we believe that ADM standards will be widely used in a near future because is an OMG initiative.

We also argue, although there are a number of other tools based on more sophisticated techniques than string analysis, our combined technique obtained reasonable values in terms of precision and recall according to our empirical evaluation. In the future we plan to improve our mining technique to identify other type of concerns such as architectural and business patterns in KDM models. Thus, it could be interesting determine possible relations between concerns belonging to high layer levels with concerns belonging to low layer levels for better comprehension of software systems.

Acknowledgments

Daniel Santibáñez would like to thank the financial support provide by CAPES. Rafael Durelli would like to thank the financial support provided by FAPESP, 2012/05168-4. Valter Camargo would like to thank FAPESP, 2012/00494-0.

References

[1] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Commun. ACM*, vol. 54, pp. 142–151, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1924421.1924451

[2] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[3] L. Mainetti, R. Paiano, and A. Pandurino, "Migros: A model-driven transformation approach of the user experience of legacy applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7387 LNCS, pp. 490–493, 2012, cited By (since 1996) 0.

[4] W. M. Ulrich and P. Newcomb, *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[5] M. von Detten, M. Meyer, and D. Travkin, "Reverse engineering with the reclipse tool suite," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 299–300. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810360

[6] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 3:1–3:37, December 2007. [Online]. Available: http://doi.acm.org/10.1145/1314493.1314496

[7] R. Durelli, D. M. Santibáñez, N. Anquetil, M. E. Delamaro, and V. V. Camargo, "A Systematic Review on Mining Techniques for Crosscutting Concerns." Coimbra, Portugal: ACM SAC, 2013.

[8] G. Deltombe, O. L. Goaer, and F. Barbier, "Bridging kdm and astm for model-driven software modernization." in *SEKE*. Knowledge Systems Institute Graduate School, 2012, pp. 517–524.

[9] R. PÃ©rez-Castillo, I. De GuzmÃ¡n, D. Caivano, and M. Piattini, "Database schema elicitation to modernize relational databases," vol. 1 DISI, no. AIDSS/-, 2012, pp. 126–132, cited By (since 1996) 0.

[10] R. Perez-Castillo, I. G.-R. de Guzman, O. Avila-Garcia, and M. Piattini, "On the use of adm to contextualize data on legacy source code for software modernization," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 128–132. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.20

[11] OMG, "Object Management Group (OMG) Architecture-Driven Modernisation," Disponível em: *http://www.omgwiki.org/admtf/doku.php?id=start*, 2012, (Acessado 2 de Agosto de 2012).

[12] K. Mens, A. Kellens, and J. Krinke, "Pitfalls in aspect mining," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 113–122.

[13] P. Parreira Junior, W. Mendes, V. de Camargo, R. Penteado, and H. Costa, "Mining crosscutting concerns with comscid: A rule-based customizable mining tool," in *Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En*, 2012, pp. 1–9.

[14] J. Han, *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[15] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[16] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Aspect recommendation for evolving software," in *Proceeding of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 361–370.

[17] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll, "Timna: a framework for automatically combining aspect mining analyses," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 184–193.

[18] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 1–37, 2007.

[19] Z. Danfeng, G. Yao, and C. Xiangqun, "Automated aspect recommendation through clustering-based fan-in analysis," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 278–287.

[20] S. Breu and T. Zimmermann, "Mining aspects from version history," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006.

[21] L. Lengyel, T. Levendovszky, and L. Angyal, "Identification of crosscutting constraints in metamodel-based model transformations," in *EUROCON 2009, EUROCON '09. IEEE*, 2009, pp. 359–364.

[22] P. Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards automating source-consistent uml refactorings," 2003.