KDM as the Underlying Metamodel in Architecture-Conformance Checking

Fernando Chagas
Universidade Federal de São Carlos - UFSCar
São Carlos, SP, Brazil
fnd.chagas@gmail.com

Rafael Durelli, Ricardo Terra
Universidade Federal de Lavras - UFLA
Lavras, MG, Brazil
{terra,durelli}@dcc.ufla.br

Valter Camargo
Universidade Federal de Sãoo Carlos - UFSCar
São Carlos, SP, Brazil
valter@dc.ufscar.br

ABSTRACT

There are two important artifacts in any Architecture-Conformance Checking (ACC) approach: i) the representation of the PA and ii) the representation of the CA. Many times, inside the same ACC approach, distinct meta-models are adopted for representing the PA and the CA. Besides, it is common the adoption of meta-models unsuitable for representing architectural details. This heterogeneity makes the checking algorithms complex since they must cope with instances that comply with two different meta-models or do not have proper architectural abstractions. KDM is an ISO meta-model proposed by OMG whose goal is to become the standard representation of systems in modernization tools. It is able to represent many aspects of a software system, including source code details, architectural abstractions and the dependencies between them. However, up to this moment, there is no research showing how KDM can be used in ACC approaches. Therefore we present an investigation of adopting KDM as the unique meta-model for representing PA and CA in ACC approaches. We have developed a three-steps ACC approach called ArchKDM. In the first step a DSL assists in the PA specification; in the second step an Eclipse plug-in provides the necessary support and in the last step the checking is conducted. We have also evaluate our approach using two real world systems and the results were very promising, revealing no false positives or negatives.

Keywords

Architecture-Driven Modernization, Knowledge-Discovery Metamodel, Architecture-Description Language, Architectural Reconciliation, Architectural Conformance Checking.

1. INTRODUCTION

Legacy information systems are usually characterized by demanding high maintenance costs, but at the same time, for being essential to support internal business processes. These systems cannot simply be discarded since they store a lot of valuable business knowledge over time [24]. For many years, reengineering has been a solution to this problem, although a study has shown that more than 50% of the reengineering projects fail. One of the main reasons is the lack of standardization [27], which hinders the reuse of algorithms and interoperability among reengineering/modernization tools.

As a solution, OMG has proposed the Architecture-Driven Modernization (ADM) [3], which is a model-driven alternative for reengineering processes. The most important ADM meta-model is the Knowledge Discovery Metamodel (KDM), whose 1.3 version was recognized as an ISO standard in 2012 (ISO/IEC 19560) [22]. Besides, more than 30 companies have participated in the specification of this meta-model.

The most important KDM characteristic is its completeness for representing source code details, architectural abstractions, deployment characteristics, UI, database, and also the dependencies among them. OMG intends to make KDM the most adopted meta-model in modernization tools (ADM-based modernization tools). Many OMG members are interested in finding out the advantages and disadvantages of KDM [22]. This will support them in deciding for the adoption of KDM in their modernization tools.

Architectural erosion is a well known problem of legacy systems, which is a progressive degradation of their architecture. This problem occurs when there is a gap between the Current Architecture (CA) and the Planned Architecture (PA). PA is the architecture that the system should have and preserve along its life in order to meet the intended quality attributes. It is an artifact that involves architectural elements (Layers, Components, etc.) and constraints/rules among these elements (e.g. Layer A cannot access Layer B). On the other hand, CA is a representation of the current system implementation; many times exposing differences when compared to the PA.

An alternative for solving the architectural erosion problem is conducting a modernization process focusing on architectural reconciliation, readjusting the CA towards the PA. The first and most important step of this process is called Architecture-Conformance Checking (ACC), whose goal is to reveal the architectural violations of the CA, when compared to the PA [6, 7]. In general, these violations are dynamic actions (methods calls, class instantiations, parameters passing, object instantiations, etc.) or structural dependencies (extends, implements, associations) that are not in accordance with the constraints/rules imposed by the PA.

In order to conduct ACC in an automatic way, PAs and CAs must be represented in a computable format. A diverse set of formats have been used in ACC approaches for specifying PAs and CAs [7, 8, 9, 10, 11, 12, 13], such as: proprietary

SBES '16, September 19-23, 2016, Maringá, Brazil
© 2016 ACM. ISBN 978-1-4503-4201-8/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2973839.2973851

meta-models, UML, etc. Besides, many times the same approach employ different meta-models for representing these architecture representations, increasing the complexity of the checking algorithms and impacting in the accuracy of the detection process. For example, Dependency Constraint Language (DCL) [28], SAVE [14], and LDM [25] employs proprietary models for representing the PA, and AST for the CA.

In this paper we present an investigation on how to use KDM as the underlying meta-model for representing both PA and CA in ACC approaches. The main motivations of this work are: the current demand for evidences of the suitability of KDM in supporting modernizations that involve ACC. Since ACC is a recurrent activity when conducting architectural modernizations, it is very important to provide evidences of the KDM suitability to support this process. The second motivation is to check if the use of a unique meta-model in ACC approaches for representing PA and CA, impact the accuracy of the checking process. Therefore, the following two general research questions drive this study: GRQ1 - Is it possible to reach good levels in terms of recall and precision when employing KDM as a base metamodel in ACC? GRQ2 - Does KDM provide all the suitable metaclasses for conducting ACC?

In order to raise conclusions, we have developed an ACC approach called ArchKDM. For the PA specification, our approach delivers a Domain-Specific Language (DSL) called DCL-KDM that generates a KDM instance that represents the PA. For the CA extraction, our approach delivers an Eclipse plug-in called ExtrArch that supports the mapping between PA abstractions and the system source code. After having these two artifacts, our approach is able to conduct the checking.

We have conducted an evaluation of our approach focused on verifying i) the effectiveness of extracting the CA and ii) the effectiveness in detecting the architectural violations. Two real-world systems were used in the evaluation (SIGA and LabSys) and the results were very promising, i.e., the approach was able to correctly generate the CA and to identify all architectural violations without false positives or negatives.

This paper is structured as follows: Section II explains ADM and KDM; Section III describes the ArchKDM and its support tools; Section IV reports a case study that evaluates our approach; Section V outlines related work, and Section VI makes concluding remarks and suggests future work.

2. ADM AND KDM

ADM [23] is an OMG initiative for standardizing system modernization processes. The main idea is take advantage of reverse engineering concepts, Model-Driven Architecture (MDA) principles and the KDM meta-model. A typical ADM modernization process starts by reverse engineering a system into a KDM instance, keeps on processing this instance to identify problems, proceeds by applying refactorings/transformations on this instance and finishes with the generation of the modernized system. According to Perez [23] there are several modernization scenarios that can be conducted to modernize legacy systems: Platform Migration, Language to Language Conversion and Application Improvement. The last one involves architecture reconstruction, which is the scenario we are dealing with in this paper.

KDM is a language and platform independent ISO meta-model capable of representing a complete software system. KDM can be seen as a family of meta-models and is composed of several meta-models that share the same vocabulary and terminology, facilitating the relationships among meta-classes in different abstraction levels. OMG intends to make KDM the most adopted meta-model for representing systems inside modernization tools. This will propitiates the interoperability among these tools and, consequently, can lead to higher success in modernization projects.

A schematic representation of KDM can be seen in Figure 1. It is divided into four layers (right side) that group packages (internal meta-models), where, each one concentrates on specific aspects of a software system. Thus, there are packages (meta-models) for representing a wide spectrum of system abstractions, from low level details of the source code (`Code` package) to run-time actions (`Action` package), to user interface (`UI` package), to deployment details (`Build` package), to Business Rules abstractions (`Conceptual` package), to Architectural details (`Structure` package), etc.

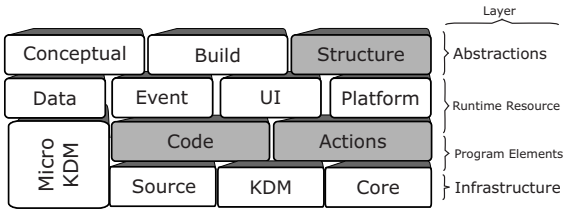

Figure 1: KDM layers

The `Code`, `Action` and `Structure` are the most important packages in the context of this work because they allow the specification of systems architectures. Figure 2 shows a small snippet of the KDM, showing some meta-classes of the Structure, Code, Core and Action packages, that is represented by the word "from [package]" under the name of each meta-class.

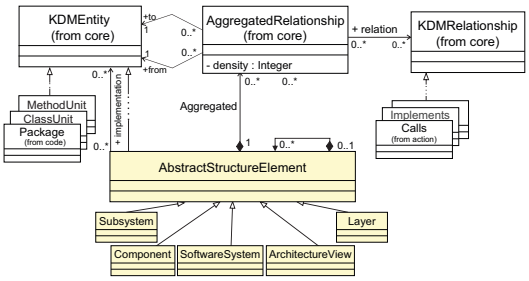

Figure 2: `Structure` Package Class Diagram (OMG Group [23])

The `Core` package is a central KDM package that provides base meta-classes for the other packages. `KDMEntity` is one of the most important meta-classes, since all the other KDM meta-classes are direct or indirect subclass of it. So, all KDM-metaclass are KDMEntities.

The `AggregatedRelationship` is also another important meta-class in this context. It is a relationship that allows to group other primitive relationships within it. This is represented by the association with the `KDMRelationship` class. In KDM, every relationship type is represented by a

meta-class, examples of primitive relationships are method calls (`Calls` meta-class), object instantiations (`Creates` meta-class), implements relationships (`Implements` meta-class), etc. Observe that each `AggregatedRelationship` involves two KDM Entities, the source and target.

The `Structure` package delivers five classes for representing architectural elements: `Subsystem`, `Component`, `SoftwareSystem`, `ArchitectureView` and `Layer`. By means of the self-relationship of the `AbstractStructureElement`, it is possible to create a hierarchy among these elements. For example, it is possible to create an architecture having two subsystems, which include two layers each, where each layer can includes two components.

Since all the architectural elements are KDM Entities (due to the inheritance), it is possible to represent relationships between these architectural elements by means of the `KDMAggregatedRelationship` (AR), which is schematically shown in Figure 3. Suppose the existence of a relationship between the layers `Controller` and `Model`. The arrow between these layers represents an instance of the `KDMAggregatedRelationship` ❷ class, where the source of the relationship is the `Controller` ❶ and the target is the `Model`. As its name suggests, an aggregated relationship incorporates primitive relationships inside itself. Primitive relationships are "actions" or structural dependencies that are also represented as KDM meta-classes.

In the figure, the AR contains one the eight instance of each primitive relationship type existent in the KDM. For example: : `calls` ❸ represents the presence of an instance of the `Calls` meta-class, showing the existence of a method call from a class that belongs to the Controller layer to a class belonging to the Model layer. The same occurs with others primitive relationships. Another important information here is that every AR has a *density*, which represents the number of primitive relationships inside it. In this example, the density ❹ is 8.

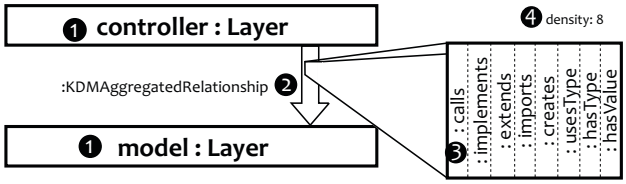

Figure 3: Schematic example of a KDM Structure Package Instance

3. ARCHKDM

3.1 Overview

As any other ACC approach, the main goal of ArchKDM is to identify violations between the PA and CA of a system. It involves three main steps depicted in Figure 4. The first is the specification of the PA; the second step is the specification of the CA and the third step is the checking; in which PA and CA are compared to detect the violations. Step II is divided into three activities. Steps **I.A.** and **II.C.** are human-dependent. The others are performed by algorithms and transformations.

It is important to highlight that the ArchKDM approach is generic, i.e., it involves the conventional steps of any ACC Approach. Besides, as the developed algorithms rely exclusively on KDM terminology, we can also say it works for checking the conformance of systems implemented in any object-oriented language. The limitation resides on the existence of parsers for different languages. Nowadays, the most mature parser is MoDisco, which is exclusive for Java. The following sub-sections detail each of the steps shown in Figure 4.

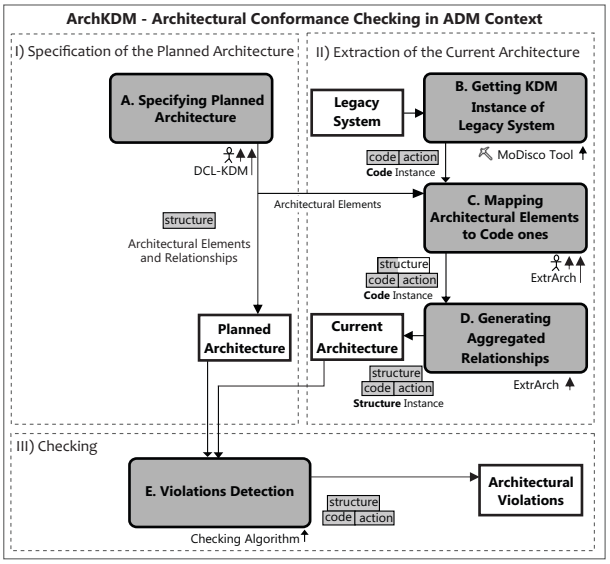

Figure 4: ArchKDM Approach

3.2 Specification of the Planned Architecture

In this Step, the goal is to create the PA by specifying all of its architectural elements (AE) and constraints among them. To support the specification we have extended an existing architecture-description language called DCL [28]. The extension comprises three points: i) the possibility of using the words: `Layer`, `Component`, `Interface`, `Subsystem` and `System` in the specification; ii) the automatic generation of some constraints and iii) the generation of the KDM instance (XML) that represents the PA. The main reasons for choosing DCL as our base **DSL** were *i)* its proximity to natural language and *ii)* the actions (method calls, instance creations, etc.) originally evaluated by DCL are perfectly supported by the KDM meta-classes.

In the original version of DCL the unique keyword (Type) used to specify AEs is `module`, which represents a set of classes. Therefore, what distinguish a type of module from the others are simply the name of them. For example, if you want to create a `Layer`, you must create a `Module` whose name indicates that. If you want to create a `Component`, you must also create a `Module` whose name indicates that. Therefore, regardless of the architectural style [26] you are using, you will always create *modules*. This way, it is very difficult to associate details that are particular to some architectural styles and is almost impossible to process them. Examples of specific characteristics to architectural styles are: levels in layers, required and provided interfaces in components and the sequence in Pipes and Filters.

As the KDM `Structure` Package has some dedicated meta-classes for some types of architectural elements, we decided to bring up these specific types as keywords in our language.

Doing that, it is possible to process each AE in a more specific way and generate some constraints (restrictions) for them. Besides, we claim that using Architectural Styles-based keywords makes the specification a more straightforward task for those familiar with architectural styles and KDM terminology. This is one of the advantages of using a meta-model that provides specific AEs when specifying PAs.

Regarding the automatic generation of constraints/restrictions, up to this moment, we have just being able to automatically generate constraints for strict layered systems and hierarchies. For example, when the software engineer specifies layers he/she needs to inform the "level" of the layer. Doing that, DCL-KDM is able to automatically generate the constraints among the layers. This is possible because in the Strict Layering Architectural Style is known that lower level layers cannot access higher level ones. In the case of hierarchies, we assume higher-level AEs can access all the services of the inner AEs. However, the inner elements cannot access the higher level ones. For example, if there is a `Component` C inside a `Layer` L, all the functionalities of the `Component` can be accessed by the `Layer`. It is important to stress that for any other case, the restrictions can also be written by the software engineer.

When specifying PAs, two issues are important: the architectural elements and the constraints between them. As KDM only provides direct support for the first one, we had to decide how to represent the constraints on it. Our decision was representing the constraints considering the presence and absence of primitive relationships inside an AggregatedRelationship. The existence indicates they are allowed and the absence indicates the opposite. In other words, if there are none relationships between layer A and B, it means that elements inside A cannot access the elements in B regardless of the way. The whole set of primitive relationships provided by KDM is shown in Figure 3. As can be seen, there are eight relationship types. Therefore, in a PA, if there is only the type `Calls` between two AEs, that means all the other seven types are not allowed between them.

Suppose the Structure instance shown in Figure 3 is a PA. This means the `Layer` Controller can access the `Layer` Model because there is an Aggregated Relationship between them. The allowed primitive relationships between these layers are those shown inside the Aggregated Relationship, i.e., all the possible types. However, if the Aggregated had only two types of relationships inside it (letÂt's say Calls and Implements), that would mean all the other types are not allowed between these layers. In the same line of thought, if there was no Aggregated between two AEs, that would mean these elements should not communicate at all.

In the next sub-sections we provide more details about each of the reserved words of our DCL-KDM and examples of their use for PA specification.

```
1 layer l1, level 1 inSubSystem main
2 layer l2, level 2 inSubSystem main
3 layer l3, level 3 inSubSystem main
```

Listing 1: Example of Layered Style Architecture in DCL-KDM

1) Layer: layer is a group of modules that offer a cohesive set of services to other layers [8]. Layers are related to each other by the strictly ordered relation allowed by their use. In addition, the relations must be unidirectional. In the context of our approach, layers are represented in DCL-KDM though the keyword `layer`, as can be seen in Listing 1.

The expression for specifying a layer is divided into two mandatory parts and one optional. Firstly, one should specify the layer via the keyword `layer`, followed by its name. Secondly, the architect must specify its level through the keyword `level`. DCL-KDM uses `level` keyword to automatically generate the architectural constraints.

The optional part is used when a layer is contained in a subsystem, component or another layer, i.e., the architect should specify in which subsystem or component the layer belongs to. Listing 2 depicts the constraints that are automatically created by our approach accordingly to the Listing 1.

```
1 l1 cannot-depend l2
2 l2 cannot-depend l3
3 l1 cannot-depend l3
4 l3 cannot-depend l1
```

Listing 2: Example of constrains automatically generated

3.2.1 Component and Interface

Component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [9]. Components have interfaces that defines a specific point of potential interaction by a component with its environment. In order to use DCL-KDM, the architect must first write the keyword `component` followed by its name, as can be seen in Listing 3. In addition, if necessary the architect can set the layer, subsystem or another component that the component belongs to. However, as previously mentioned, components require auxiliary structures to represent constraints, these elements are called interfaces.

An interface is declared by the `interface` keyword, followed by its name. Its specification also requires that the architect defines the component which it belongs to. Unlike layers, the architectural style defined by components requires that its constraints be explicitly specified. Listing 3 show as the component and interfaces are specified and how they interact.

```
1 component c1;
2 component c2;
3 interface i_c1 ofComponent c1;
4 c2 can-depend-only i_c1;
```

Listing 3: DCL-KDM code to define interfaces and its constraints

Lines 1-2 specify the components. Line 3 describes an interface named `i_c1` that is provided to the component `c1`. An architectural constraint is defined in line 4. It states that component `c2` can depend only of `i_c1` interface.

3.2.2 Subsystem

A subsystem may be independently implemented by any other AE [8]. The requirements are: (*i*) Under subsystem must have the `subSystem` keyword, followed by its name, and (*ii*) it may be in another subsystem. An important difference between a subsystem (DCL-KDM) and a module (DCL) is the hierarchy position of subsystems in an architecture, because DCL-KDM predicts that a subsystem can comprise of: layer, components, and subsystems.

3.2.3 Software System and Module

All rules outlined in our approach consider that, by default, there is a root element in the architectural description hierarchy, which is called `SoftwareSystem`. Thus, it is automatically added to KDM `Structure` model. We have decided to not eliminate the `Module` element, it deals with cases of explicit rules, in the same way that it is used in DCL.

As long as the software engineer has specified the PA with DCL-KDM, the PA is generated as an instance of the KDM Structure package. The output of this step is a KDM instance just with the `Structure` package instantiated, which represents the PA, i.e., the `Code` and `Action` packages are totally empty.

An example of a PA specification using DCL-KDM is shown in Listing 4. This example is related to the PA of the LabSys system, which we have used in our examples and evaluation.

```
1 architeturalElements {
2     subSystem core;
3     layer view, level 3, inSubSystem: core;
4     layer controller, level 2, inSubSystem: core;
5     interface consumerInterface ofComponent consumer;
6     layer model, level 1, inSubSystem: core;
7     module repository, inLayer: model;
8     component generic;
9     interface genericInterface ofComponent generic;
10    component converter;
11    interface converterInterface ofComponent converter;
12    module validator;
13 } restrictions {
14    repository can-depend-only controller;
15    only controller can-depend repository;
16    only controller can-depend validator;
17    genericInterface can-depend-only controller;
18    converterInterface can-depend-only controller;
19 }
```

Listing 4: Planned Architecture of LabSys

The specification is divided in two parts. The first one (lines 2-12) describes the AEs and the hierarchy among them. The second one (lines 13-18) describes the restrictions among the AEs. Notice that some restrictions are automatically generated and are not shown in the Listing. Lines 2-12 show the AEs and hierarchy between them. The hierarchy automatically generates its constraints, avoiding the need for creating restrictions in theses cases. Lines 13-18 show the constraints. Note that no restriction was defined between layers `model`, `view`, and `controller`, because they are automatically generated by DCL-KDM.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <kdm:Segment xmi:version="2.0" [...]
3  xmlns:kdm="http://kdm.omg.org/kdm"
4  xmlns:structure="http://kdm.omg.org/structure">
5  <model xsi:type="structure:StructureModel" name="LabSys">
6    <structureElement xsi:type="structure:SoftwareSystem" name="LabSys">
7      <structureElement xsi:type="structure:Subsystem" name="core">
8        <structureElement xsi:type="structure:Layer" name="controller">
9          <aggregated from="//@model.0/[...]/@structureElement.0"
10         to="//@model.0/[...]/@structureElement.1"
11         relation="//@model.1/[...]/@actionRelation.0
12         //@model.1/[...]/@codeElement.0/@actionRelation.1 //
13         [...]//@model.1/@codeElement.0/@codeRelation.2" density="8"/>
14       </structureElement>
15     <structureElement xsi:type="structure:Layer" name="model"/>
16     <structureElement xsi:type="structure:Layer" name="view">
17       <aggregated from="//@model.0/[...]/@structureElement.2"
18       to="//@model.0/@structureElement.0/[...]/@structureElement.0"
19       relation="//@model.1/[...]/@actionRelation.0
20       //@model.1/[...]/@codeElement.0/@actionRelation.1 //
21       [...]//@model.1/@codeElement.0/@codeRelation.2" density="8"/>
22     </structureElement>
23   </structureElement>
24  </structureElement>
25 </model>
26 [...]
27 </kdm:Segment>
```

Listing 5: A KDM Instance representing the PA of the LabySys

In Listing 5 it is shown a XML KDM instance generated by DCL-KDM and that represents a snippet of the PA specification shown in Listing 4. Line 5 shows the `StructureModel`, which defines a new architectural specification. Lines 8, 15 and 16 show instances of the `Layer` meta-class representing the layers model, view and controller. The allowed relationships between these layers are represented by instances of the `AggregatedRelationship` class, as presented in lines 9-13 and 17-21.

It is important to stress that each of these instances is composed by a source element (line 9), the target one (line 10) and the primitive relationships, which are not shown because of space limitations. The density is defined (line 13), that indicates the quantity of relationships accepted between the source and target elements.

3.3 Extraction of the Current Architecture

This subsection provides an overview of the process for extracting the CA, which is another KDM instance that represents this architecture. This step involves three activities shown in Figure 4 and it is supported by an Eclipse Plug-in we have developed called ExtrArch. The three activities are briefly commented below.

In activity **II.B**, **Getting KDM Instance of Legacy System**, the MoDisco tool [6] is employed to obtain an initial KDM instance that represents all source code elements (classes, actions, attributes, relationships, etc.). This KDM instance is incomplete, since only the `code` and `action` packages are instantiated by MoDisco.

In activity **II.C**, **Mapping Architectural Elements to Code Ones**, a mapping between the architectural elements (declared in the PA specification - activity **I.A.**) and the code elements (collected from MoDisco - activity **II.B.**) must be created by the software engineer. The intention is to inform that specific code elements are the implementation of some AEs. For example, a package P1 is the implementation of a layer L1. The mapping is supported by the ExtrArch plug-in, in which the engineer chooses an architectural element on the left side and assigns it to code elements on the right side.

When the mapping is not clear, the process must be supported by system specialists to reach an optimal mapping, which is generally possible considering the granularity of our architectural elements. In terms of our tool support, the engineer can iterate on the process by creating several mapping versions that can be analyzed later. It is significant to highlight that our approach allows a low level of granularity, i.e., in addition to `package`Âť's, our approach also support classes and interfaces. At the end of this activity, the output is a KDM instance that represents a partial CA. It is called "partial" because it is still missing the relationships (aggregated relationships) among the architectural elements, which are the most important information for detecting the violations. These relationships are automatically generated by the next activity.

In activity **II.D.**, **Generating the Aggregated Relationships**, an algorithm scans all code elements in the current KDM instance collecting all the actions and structural dependencies between these elements. Based on that, the algorithm is able to generate the aggregated relationships between the architectural elements. For example, suppose the previous step has defined the package `P1` is the implementation of the Layer `L1` and package `P2` is the implementation of the Layer `L2`. If there is a method call from a class of `P1` to a class of `P2`, then the algorithm creates an aggregated relationship between the layers `L1` and `L2` and insert this primitive relationship inside it. Besides, it also calculates the density value.

Algorithm 1: ExtrArch - Extracting Algorithm

```
   Input: KDM Instance with code and action packages fully
          instantiated and structure package partially.
   Output: KDM Instance with code, action, and structure
           packages fully instantiated (current system
           architecture).
1  begin
2      pRelationship =
         kdmUtil.getAllPrimitiveRelationships(currentArq)
3      foreach primRelationships pRelationship do
4          sourceElement = pRelationship.getSource()
5          targetElement = pRelationship.getTarget()
6          aElementSource =
             getArchitecturalElement(sourceElement)
7          aElementTarget =
             getArchitecturalElement(targetElement)
8          foreach aElementSource.aggregatedRelatioships
             aggregatedRelSource do
9              if aggregatedRelSource target equals to then
10                 auxAgRelationship =
                     aElementSource.getAggregatedRelatioship
11                 foreach outgoing auxAgRelationship do
12                     if auxAgRelationship.getTo =
                         aElementTarget then
13                         auxAgRelationship.add(pRelationship)
14                         auxAgRelationship.density(aElement-
15                         Source.getAggregatedRelatioship
16                         .getDensity+1)
17                     end
18                     if auxAgRelationship is the last then
19                         auxAgRelationship = new
                             AggregatedRelatioship
20                         auxAgRelationship.add(pRelationship)
21                         auxAgRelationship.density(1)
22                     end
23                 end
24             else
25                 auxAgRelationship = new
                     AggregatedRelatioship
26                 auxAgRelationship.add(primitiveRelationship)
27                 auxAgRelationship.density(1)
28             end
29         end
30     end
31 end
```

Algorithm 1 shows the code responsible for extracting CA, which is called "extracting algorithm". The first step of the algorithm is to recover all primitive relationships (method calls, implementation of interfaces, etc) (line 2). This part was implemented by an auxiliary algorithm that performs a depth-first search in KDM-tree. Then, for each meta-class representing a primitive relationship, its respective source code elements (method, class, package, etc.) and targets are searched (lines 3-4). Next, the architectural elements of source (layer , component, subsystem) and target are searched (lines 5-6).

Additionally, it is verified if any architectural level relationship exists (`AggregatedRelationship`) in the source architectural element (line 7). If so, it should be checked if the destination is the same as a searched relationship (line 11), then the primitive relationship should be inserted (lines 12-14), otherwise, a new one should be created (line 17-19). Finally, if the source architectural element still does not have an `AggregatedRelationship`, a new one should be created (lines 22-24). Thus, each one of the primitive relationships are added to their respective `AggregatedRelationship`, without creating any unnecessary relationship. The output of this step is a KDM instance representing the CA of the system, which will be compared to the PA generated in the previous step. Both are XMI documents, as shown in Listing 5.

3.4 Checking

In this subsection we present the Conformance Checking, as shown in Figure 4, Step III. This step is supported by our Checking Algorithm shown in Algorithm 2. The input are two KDM instances (PA and CA) generated for the DCL-KDM and ExtrArch, respectively. The algorithm compares these two KDM instances, checking which relationships of the CA do not exist in the PA, characterizing the possibility of an architectural violation. The output is a KDM instance containing all violations possibilities.

Algorithm 2: ArchKDM - Checking Algorithm

```
   Input: KDM instances that represents the planned plannedArq
          and current currentArq architectures.
   Output: KDM instance containing the architectural violations
           violationsArq.
1  begin
2      aggregatedRelCA =
         kdmUtil.getAllAggregatedRelationships(currentArq)
         aggregatedRelPA =
         kdmUtil.getAllAggregatedRelationships(plannedArq)
3      foreach aggregatedRel aggregatedRelCA do
4          plannedRelationship =
             seekCorrespondentRelationship(aggregatedRel,
             plannedArq)
5          if plannedRelationship is empty then
6              violationsArq.addViolation(aggregatedRel)
7          end
8          else
9              primitiveRelationshipsCA = kdmU-
                 til.getAllPrimitiveRelationships(aggregatedRel)
                 primitiveRelationshipsPA = kdmU-
                 til.getAllPrimitiveRelationships(aggregatedRel)
                 foreach primitiveRel primitiveRelationshipsCA
                 do
10                   if primitiveRel not exist in
                       primitiveRelationshipsPA then
11                       violationsArq.addViolation(primitiveRel)
12                   end
13             end
14         end
15     end
16 end
```

The first part of this algorithm is recover all architectural level relationships of the PA and CA representations (line 2). Then, the list of CA relationships is iterated (lines 3-15). Next, the algorithm checks which PA relationships has the same source and target of the CA relation (line 4). Then is verified if the PA relationship does not have any relationship, in positive case, a new one is added (lines 5-6). In negative case, the relationships that are not present in the PA are included in a list of relationships that are not allowed (lines 9-12). Therefore, at the end of the algorithm execution, it must exist a KDM instance that has the list of violations.

The working of ArchKDM can also be explained using Set Theory. Figure 5 shows an example. Let R be the Set of whole spectrum of possible relationships provided by KDM, i.e., the eight primitive relationships shown in Figure 4. On one hand, let PA be the Set of the relationships specified by the software engineer as the allowed relationships among AEs. PA is a subset of R, i.e., $PA \subset R$. Consider a PA composed by two relationships, a Calls and an Implements, as shown in Figure 5. On the other hand, let CA be the Set of the relationships of the current architecture of the system under analysis. CA is also a subset of R, i.e., $CA \subset R$. A software system in full conformance with its architectural design indicates $CA \subseteq PA$. The set V of the existing architectural violations is the difference between CA and PA, i.e.,

CA - PA. So, in the example shown below, $CA - PA = \mathrm{V} = \{Extends\}$.

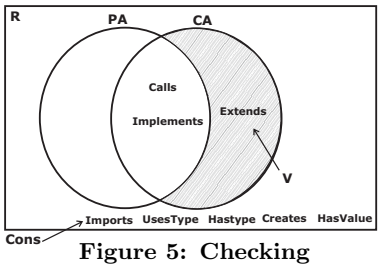

Figure 5: Checking

4. EVALUATION

The best possible result of our approach is to detect all the existing architectural violations in a given system, without false positives or negatives. However, the effectiveness of the whole approach depends on three main steps: i) the correct specification and generation of the PA (Step **I**); ii) the correct extraction of the CA (Step **II**); and iii) the quality of the checking algorithm. In this paper we decided to evaluate only non-human-dependent activities. Therefore, the evaluation concentrated on the extracting (Activity **II.D** - Figure 4) and the checking algorithms (Step **III** - Figure 4).

4.1 Definition of the Empirical Study

Goal: The general goal is to evaluate the effectiveness of the approach in detecting architectural violations. Therefore, we divided the evaluation into two parts: i) evaluation of the extracting algorithm (Activity **II.D** - Figure 4), which recovers the CA, and ii) evaluation of the checking algorithm (Step **III** - Figure 4), which identifies the architectural violations between PA and CA.

Quantitative Approach: The potential of the approach is measured by means of the percentage of correct relationships (CA) and violations (architecture conformance checking) that the algorithms are able to detect.

Perspective: Software architects.

Study Object: The relationships in the extracted CA and the violations detected by the conformance checking.

4.2 Study Planning

In order to answer the General Research Question 1 shown in Section 1 (Is it possible to reach good levels in terms of recall and precision when employing KDM as a base metamodel in ACC?) we decided to break it into two more specific ones: 1) Assuming that the software engineer has provided a correct mapping in Step **II**, is the extracting algorithm able to recover the correct CA? 2) Assuming that the PA and CA specifications are correct, is the checking algorithm able to detect all the architectural violations? Therefore, to answer these research questions, we collected the following data:

- The relationships among code elements (classes/interfaces) that are automatically extracted by our tool;
- The violations detected by our checking algorithm.

The focus of Item 1 is to verify if the extraction algorithm is able to generate all the relationships presented in the source code, for a given system. The focus of Item 2 is to check if the checking algorithm is able to identify violations between all types of elements (layers and components, layers and systems, layers and modules, etc.) and constraints (calls, extends, etc.).

4.2.1 Context Selection

We have used two real-world systems in our evaluation: LabSys (Laboratory System) and SIGA (Integrated System of Academic Management). LabSys is a laboratory management system currently used by the Federal University of Tocantins (UFT) and SIGA is an academic management system currently used by the Federal University of São Carlos (UFSCar). We chose LabSys because it involves a considerable variety of combinations among architectural elements. We chose SIGA since it is a large system, and its software engineers are very interested in detecting the system architectural violations.

4.3 Operation

4.3.1 Preparation

We have built two oracles for LabSys; one containing all architectural violations (called violations oracle) and another for its CA, containing all relationships among its architectural elements. That was possible because an author of this paper was one of the main developers of this system. So, he knows the system in a deep way. Even considering his experience, the building of the oracles was an intensive work for several days.

For building the oracles, each element of each class was manually scanned and compared to the PA and a list of them was elaborated. This process took five weeks and it was conducted by three people; a master degree student that was the main developer, the software engineer who is currently the responsible for the LabSys and a phD student of our lab. In the case of SIGA, the PA was provided by the project manager from the Development Sector of UFSCar. However, the building of an oracle was impractical because of its size. Thus, we just verified if the architectural violations found by our tool were true positives, i.e., they actually refer to design decisions that are not prescribed by the PA.

4.3.2 Execution

We performed the approach following all ArchKDM's steps. Initially, we used DCL-KDM to specify the PA. Next, we used the ExtrArch to map the architectural elements of the PA to the source code elements. Finally, we triggered the violation detection.

4.3.3 Data

We extracted all relationships and violations to compare them with LabSys oracles.

We analyzed architectural violations from three perspectives: amount of violations, constraints between different elements and different types of restrictions. The amount of violations are used to verify if the checking was successful. Then, we checked if all possible architectural violations between two elements of different types were found, such as those found between a layer and a subsystem, or between a layer and a component. Finally, we checked if all kinds of restrictions were found, e.g., method calls, variables accesses, and inheritance.

4.4 Analysis of the Current Architecture Extraction Process

The goal here is to determine if the extracting algorithm is able to generate the CA correctly. The critical part of this step is the extraction of existing relationships, since an error in the underlying algorithm may lead to incorrect results. Table 1 list the LabSys relationships in combination with the oracle results. It is important to remember that every existing relationship in the source code is an instance of a specific meta-class. For example, looking at the third line of the table, it is possible to see that the algorithm generated 240 instances of the `Calls` meta-class, between the `Controller` and `Model` layers.

Regarding the result, all relationships found by the tool were also present in the oracle. Thus, the extracting algorithm is able to correctly detect all relationships. Another relevant point is that the algorithm properly retrieved the eight kinds of relationships (calls, extends, etc.) determined in the proposed approach.

4.5 Analysis of the Checking Algorithm

The goal here is to evaluate the amount of architectural violations detected by the checking algorithm. Table 2 presents the results, showing the application name, the amount of architectural violations found (Architectural Violations - AV), the amount of architectural violations presented in the oracle (Architectural Violations Oracle - AVO), the percentage of false positives (FP), and the amount of false negatives (FN). Regarding the SIGA system, we have not built an oracle, therefore AVO and FN are empty.

The results indicate that we found 43 architectural violation indications in LabSys. Thus, ArchKDM could find all violations identified by the oracle, thus, no false negatives were found. For the SIGA application, we checked with the software engineers if the architectural violation evidences found by ArchKDM were correct. The tool did not return any false positives.

Despite the good levels achieved after analysis of the results regarding the amount of false positives and negatives, some variables still need to be exercised so that the quality checking can be evaluated more accurately. Thus, another point we have evaluated is the ability of the algorithm in detecting violations between all architectural element types. For example, between Layers and Components, Layers and Subsystems, Layers and Layers, etc. Table 3 reports the results. In LabSys, only the combination of subsystem-subsystem was not evaluated because there are not two subsystems in the architecture. However, in SIGA application, we found architectural violations between all combinations. Therefore, ArchKDM was able to detect deviations between all combinations.

4.6 Threats to Validity

We must state at least two threats of the reported evaluation. First, even though we rely on two real-world systems that have different architecture and constraints, we cannot claim that our approach will provide equivalent accuracy rates in other systems, as it usually happens in empirical studies of software engineering (*external validity*). Second, we relied on two software engineers to evaluate the amount of false positives. As typical in human-based classifications, our results might be affected by some degree of subjectivity (*construct validity*). However, it is important to highlight that we interviewed the software engineers who designed the architecture, and whom are responsible for their maintenance and evolution.

5. RELATED WORK

Researchers have been proposing Architectural Conformance Checking approaches based on several underlying models, in which we divided in the following four groups: i) AST-based approaches; ii) Graph-based ACC approaches; iii) MDE-based approaches and iv) other approaches.

AST-based ACC approaches: DCL [28], ArchJava [4], and ArchLint [20] rely on AST (Abstract Syntax Tree) as the underlying model for performing ACC. DCL [28] employs static analysis for identifying the structural dependencies that does not respect the rules specified in the PA. ArchJava [4] extends Java with architectural modeling constructs that seamlessly unify software architecture with implementation, ensuring that the implementation is according to the architectural constraints. ArchLint [20] is a data mining approach for ACC that identifies architectural violations based on a combination of static and historical source code analysis that frees architects from specifying the architectural constraints. These three studies share the same weakness. Although they achieve good levels in ACC, they do not support multiple languages, architectural styles, and explicitly hierarchy between the architectural elements as our approach does.

Graph-based ACC approaches: ConQAT [12], SAVE [17, 14], and SotoArch [15] rely on graphs as the underlying model to perform ACC. ConQAT [12] identifies divergences and absences based on the comparison between a machine readable specification of the intended architecture and the knowledge of the dependencies extracted automatically from the source code. Based on pure reflexion model concepts, SAVE [17, 14] highlights convergent, divergent, and absent relationships between the high-level model and the source-code model that are also automatically extracted from the source code. SotoArc [15] provides means to visualize and understand the static structure of a software system, including modeling the intended architecture and detecting architectural violations. Although complete and accurate, these tools rely on proprietary models to represent the intended architecture. Our approach, on the other hand, relies on an ISO meta-model (KDM) to represent the PA and CA. It means that researchers who are familiar to KDM can develop and improve any of our approach steps, e.g., implementing a more sophisticated CA extraction algorithm or performing high-level refactorings for the identified violations.

MDE-based ACC approaches: ArchConf [1], ReflexML [2], and Herold and Rausch's approach [16] rely on MDE models to perform ACC. ArchConf [1] generates a conformance view and computes metrics between two C&C (component and connector) views. They uniformly represent various languages of the system in the form of a meta-model of the relevant source artifacts at the desired level of detail. ReflexML [2] defines the traceability of UML component models to code using AOP type pattern expressions. Herold and Rausch [16] express architectural rules as formulas on a common ontology, and models are mapped to instances of that ontology. A knowledge representation and reasoning system is then used to check whether the architectural rules are satisfied for a given set of models. Although MDE-based approaches promote reuse, they do not accurately represent implementation details. Our approach, however,

Table 1: Recovered relationships of the CA LabSys

Architectural Element 1	Architectural Element 2	Calls	Implements	Extends	Imports	Creates	UsesType	HasType	HasValue
		A/O	A/O	A/O	A/O	A/O	A/O	A/O	A/O
view - Layer	controller - Layer								
view - Layer	model - Layer								
controller - Layer	model - Layer	240/240		21/21	52/52		19/19	150/150	
view - Layer	repository - Module								
controller - Layer	repository - Module	208/208			24/24			11/11	
model - Layer	repository - Module	6/6		24/24	37/37	1/1	2/2	128/128	3/3
view - Layer	generic - Component								
controller - Layer	generic - Component	108/108			8/8				
model - Layer	generic - Component								
repository - Module	generic - Component				25/25				
view - Layer	validator - Module								
controller - Layer	validator - Module	16/16			8/8			2/2	
model - Layer	validator - Module	9/9	8/8		8/8	1/1	1/1	9/9	1/1
repository - Module	validator - Module				12/12				
generic - Component	validator - Module								
view - Layer	converter - Component								
controller - Layer	converter - Component				1/1				
model - Layer	converter - Component	2/2	12/12		12/12	1/1			
repository - Module	converter - Component	1/1			1/1				
generic - Component	converter - Component								
validator - Module	converter - Component								

A = Algorithm, O = Oracle

Table 2: Violations in LabSys and SIGA

Application	AV	AVO	FP	FN
LabSys	43	43	0	0
SIGA	115	-	0	-

Table 3: Violations between different elements

Element 1	Element 2	LabSys	SIGA
Layer	Layer	Yes	Yes
Layer	Subsystem	Yes	Yes
Layer	Component	Yes	Yes
Layer	Module	Yes	Yes
Subsystem	Component	Yes	Yes
Subsystem	Module	Yes	Yes
Module	Component	Yes	Yes
Subsystem	Subsystem	No	Yes
Module	Module	Yes	Yes
Component	Component	Yes	Yes

relies on KDM, which provides a complete specification of architectural elements and allows source code elements to be represented with one-to-one precision.

Other ACC approaches: LDM [25] relies on Dependency Structure Matrices (DSMs) to perform ACC. A DSM is a weighted square matrix whose both rows and columns denote classes from an object-oriented system and the number of references that `B` contains to `A` is represented in cell (`A`, `B`). Although DSM is important for documentation purposes and communication with stakeholders, DSM is not an architecture specification that is independent of the systemâĂŹs implementation. In the dynamic analysis research line, DiscoTect [29] dynamically monitors a running system to derive its software architecture. Thus, architects can develop mappings to exploit regularities in the system implementation and architectural styles. Similarly, ConArch [7] is a runtime verification approach for detecting inconsistencies between the dynamic behavior of the documented architecture and the actual runtime behavior of the system. However, these studies share the same problem: mappings between low-level system observations and architectural events are not usually one-to-one and hence it is not straightforward to indicate implementation patterns that represent the target architecture.

6. CONCLUSION

Our approach can be used for checking the conformance of systems implemented in any language. This is possible because all the algorithms are dependent only on the KDM terminology. Besides, as our algorithms are totally based on an ISO pattern, they have a great potential for reuse. This does not happen when algorithms are developed over a proprietary or language-dependent model. Summing up, the main advantage of our approach is that it performs all the activities over an ISO platform and language-independent meta-model, and not over proprietary one. KDM is implemented by the MoDisco tool, which is a good reverse engineering tool. So, to use our approach one needs to convert their systems into a KDM representation using MoDisco and install our plug-in. We believe the use of KDM is what makes our approach attractive and unique.

Considering the GRQ2 shown in Section 1, an important discussion is, an important discussion here is regarding the suitability of the KDM for representing software architecture. The code package is clearly able of representing a very good/low level of details, however, the quality of the `Structure` package is more difficult to evaluate. For example, although the `Structure` package has the most conventional meta-classes for representing architectural details, it lacks of some other important ones, for example: Filters, Connectors, Ports, Required and Provided Interfaces, etc. In our case studies these classes were not necessary. It is worth to note that we did not extend KDM to represent architectural details, e.g., ports and connectors. Thus, we were restricted to the abstractions provided by KDM.

In the last step of our approach, both representations are compared and a list of architectural violations are obtained. As both system representations are instances of the same meta-model, the algorithm (see Algorithm 2) becomes clearer, easier to understand and, as a consequence, easier to maintain, reuse, and evolve. To conclude, based on our evaluation, the ACC was applied successfully. The usage of KDM does not impact the process quality, mainly because the `Code` meta-model is in an abstraction level very similar to the source code, thus, every detail to perform an architectural check, such as dynamic code actions (calls, instantiations, etc.) are available. Although checking for relationships is the unique type of ACC we cover, this is also the most common

type of deviations existent in software systems. We claim the deviation types we have approached represent a significant portion of all architectural deviation that occur in reality.

Another important point to note is that the advantage of KDM is its standardization. Thus, modernization tools now have good reasons for adopting KDM as the main underlying meta-model. The reason is that there probably will be a lot of resources (e.g., ACC/refactoring algorithms/tools/techniques) available that can be reused [28, 29]. Our algorithms are reusable across KDM-compliance tools because their source-code only mentions the original names of KDM meta-classes. To sum up, there is good evidence that its perfectly possible to conduct ACC in the ADM context and to obtain good results in terms of violation detection.

Acknowledgments

We would like to thank CPNQ and Fapesp (2012/05168-4).

7. REFERENCES

[1] M. Abi-Antoun and J. Aldrich. Tool support for the static extraction of sound hierarchical representations of runtime object graphs. In *23rd Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, pages 743–744, 2008.

[2] J. Adersberger and M. Philippsen. ReflexML: UML-based architecture-to-code traceability and consistency checking. In *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 344–359. 2011.

[3] ADM. Architecture-Driven Modernization. Available on: http://adm.omg.org/, 2015.

[4] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *24rd International Conference on Software Engineering (ICSE)*, pages 187–197, 2002.

[5] R. A. Bittencourt, G. J. de Souza Santos, D. D. S. Guerrero, and G. C. Murphy. Improving automated mapping in reflexion models using information retrieval techniques. In *17th Working Conference on Reverse Engineering (WCRE)*, pages 163–172, 2010.

[6] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. MoDisco: A generic and extensible framework for model driven reverse engineering. In *25th International Conference on Automated Software Engineering (ASE)*, pages 173–174, 2010.

[7] S. Ciraci, H. Sozer, and B. Tekinerdogan. An approach for detecting inconsistencies between behavioral models of the software architecture and the code. In *36th Annual Computer Software and Applications Conference (COMPSAC)*, pages 257–266, 2012.

[8] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[9] B. Councill and G. T. Heineman. Definition of a software component and its elements. *Component-based software engineering: putting the pieces together*, pages 5–19, 2001.

[10] E. Dashofy, H. Asuncion, S. Hendrickson, G. Suryanarayana, J. Georgas, and R. Taylor. ArchStudio 4: An architecture-based meta-modeling environment. In *29th International Conference on Software Engineering (ICSE)*, pages 67–68, 2007.

[11] L. de Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[12] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with conqat. In *32nd International Conference on Software Engineering (ICSE)*, pages 247–250, 2010.

[13] R. S. Durelli, D. S. Santibanez, M. E. Delamaro, and V. V. de Camargo. Towards a refactoring catalogue for knowledge discovery metamodel. In *15th International Conference on Information Reuse and Integration (IRI)*, pages 569–576, 2014.

[14] S. Duszynski, J. Knodel, and M. Lindvall. SAVE: Software architecture visualization and evaluation. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 323–324, 2009.

[15] hello2morrow. Sotoarc, 2015.

[16] S. Herold and A. Rausch. Complementing model-driven development for the detection of software architecture erosion. In *5th International Workshop on Modeling in Software Engineering (MiSE)*, pages 24–30, 2013.

[17] J. Knodel, D. Muthig, M. Naab, and M. Lindvall. Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294, 2006.

[18] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 1–12, 2007.

[19] C. Maffort, M. T. Valente, M. Bigonha, N. Anquetil, and A. Hora. Heuristics for discovering architectural violations. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 222–231, 2013.

[20] C. Maffort, M. T. Valente, R. Terra, M. Bigonha, N. Anquetil, and A. Hora. Mining architectural violations from version history. *Empirical Software Engineering*, 21(3):854–895, 2016.

[21] D. S. Martín Santibáñez, R. S. Durelli, and V. V. de Camargo. A combined approach for concern identification in kdm models. *Journal of the Brazilian Computer Society*, 21(1):1–20, 2015.

[22] OMG. Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM). Available on: http://www.omg.org/spec/KDM/1.3/, 2016.

[23] 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 *16th Working Conference on Reverse Engineering (WCRE)*, pages 128–132, 2009.

[24] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini. Knowledge discovery metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Comput. Stand. Interfaces*, 33(6):519–532, 2011.

[25] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *20th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176, 2005.

[26] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

[27] H. Sneed. Estimating the costs of a reengineering project. In *12th Working Conference on Reverse Engineering (WCRE)*, pages 9–119, 2005.

[28] R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[29] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A system for discovering architectures from running systems. In *26th International Conference on Software Engineering (ICSE)*, pages 470–479, 2004.