Supporting the Specification and Serialization of Planned Architectures in Architecture-Driven Modernization Context

André de S. Landi
Federal Univ. of São Carlos
São Carlos, SP, Brazil
andre.landi@dc.ufscar.br

Fernando Chagas
Federal Univ. of São Carlos
São Carlos, SP, Brazil
fernando.chagas@dc.ufscar.br

Bruno M. Santos
Federal Univ. of São Carlos
São Carlos, SP, Brazil
bruno.santos@dc.ufscar.br

Renato S. Costa
Federal Univ. of São Carlos
São Carlos, SP, Brazil
renato.costa@dc.ufscar.br

Rafael Durelli
Federal Univ. of Lavras
Lavras, MG, Brazil
rafael.durelli@dcc.ufla.br

Ricardo Terra
Federal Univ. of Lavras
Lavras, MG, Brazil
terra@dcc.ufla.br

Valter V. de Camargo
Federal Univ. of São Carlos
São Carlos, SP, Brazil
valter@dc.ufscar.br

***Abstract*—Architecture-Driven Modernization (ADM) intends to standardize software reengineering by relying on a family of standard metamodels. Knowledge-Discovery Metamodel (KDM) is the main ADM ISO metamodel aiming at representing all aspects of existing legacy systems. One of the internal KDM metamodels is called Structure, responsible for representing architectural abstractions (Layers, Components and Subsystems) and their relationships. Planned Architecture (PA) is an artifact that involves not only the architectural abstractions of the system but also the access rules that must exist between them and be maintained over time. Although PAs are frequently used in Architecture-Conformance Checking processes, up to this moment, there is no contribution showing how to specify and serialize PAs in ADM-based modernization projects. Therefore, in this paper we present an approach that i) involves a DSL (Domain-Specific Language) for the specification of PAs using the Structure metamodel concepts; and ii) proposes a strategy for the serialization of PAs as a Structure metamodel instance without modifying it. We have conducted a comparison between DCL-KDM and other techniques for specifying and generating PAs. The results showed that DCL-KDM is an efficient alternative to to generate instances of the Structure metamodel as a PA and to serialize it.**

1. Introduction

Architecture-Driven Modernization (ADM) combines Model-Driven Architecture (MDA), standard metamodels and the traditional reengineering phases in an unique conceptual framework. The main idea resides in creating and delivering a set of metamodels that become standards and are adopted into modernization tools. The goal is that modernization projects can be conducted employing just ADM metamodels, resulting in an ecosystem of solutions (mining algorithms, transformation rules, etc) that recognize these metamodels. If this happens, these solutions that manipulate these metamodels instances can be interchanged/reused among these tools, promoting reusability [1], [2].

Knowledge Discovery Metamodel (KDM) is the central metamodel of the proposal aiming at representing all aspects of the legacy system to be modernized. It is able of representing a wide set of aspects, ranging from low level details (such as source code) to higher level abstractions (such as architectural concepts). Because of its broadness, KDM is organized in packages that represents different abstractions of the system. In fact, each package is an internal KDM metamodel responsible for one specific aspect of the system. The Structure package is the most important KDM package in the context of this work. It contains metaclasses for representing the logical architecture of the software (layers, components, etc) and the existing relationships between the architectural elements/abstractions [3].

A recurrent problem in legacy systems is the architectural erosion, which is gradual degradation of the system architecture. An existing technique to cope with this problem, which is recurrent in modernization projects, is the Architecture-Conformance Checking (ACC) whose goal is to detect the architectural drifts of the Current Architecture (CA) of the system when compared to the architecture the system should have (PA - Planned Architecture) [4], [5]. PA is an artifact different from a conventional architecture specification; a planned architecture specifies not only the architectural abstractions (Layers, Components, etc.) the system should have, but also the access rules that must be maintained among those elements. For example: Layer A cannot access Layer B.

In order to automate ACC processes, PAs and CAs must be serialized in physical files and organized according to a specific format/structure. Having these both representations, an algorithm performs the checking process, comparing them. A diverse set of strategies have been used in ACC approaches for specifying and serializing PAs and CAs. The majority employ Domain-Specific Languages (DSLs) for the specification and proprietary metamodels for the serialization [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. To the best of our knowledge, there is no proposal in literature addressing PAs in ADM-based modernization projects. The only existing alternative that gets closer is KDM-SDK, a general-purpose API. However, the specification becomes too verbose compromising the productivity. Besides, there is no support for specifying access rules.

According to Object Management Group (OMG), the exclusive usage of ADM metamodels in modernization projects can promote a better level of reusability and standardization [1], [2]. This happens because all the solutions that act over ADM's metamodel instances recognize the structure of these metamodels and can be more easily interchanged among modernization tools that recognize such metamodels. Besides, representing PAs as KDM instance makes the algorithm that compare the PA with the CA much clearer, as both follow the same structure/terminology. So, whenever possible, it must be avoided the employing of non-ADM metamodels [1], [2].

Therefore, we present an approach for supporting the specification and serialization of PAs to be used along an ACC process in ADM-based projects. The specification is supported by a Domain-Specific Language (DSL) called DCL-KDM, which allows the architects employing the Structure Package terminology when creating the specification. The serialization is supported by a module that serialize the specified PA as a KDM instance, without employing non-ADM metamodels.

One important feature of our approach is the automatic generation of access rules for strict layering and compositions. Therefore, architects do not need to write these rules when specifying PAs that involve these kind of architectural styles [16], [17]. Besides, we employ the original versions of KDM and Structure Package, without extending them.

Section 2 describes Structure Package of KDM; Section 3 presents our tool-supported approach; Section 4 outlines the evaluation; Section 5 presents related works and Section 6 presents the conclusion.

2. The KDM Strucure Package

KDM is the central metamodel of ADM because it allows the representation of the system to be modernized. Structure Package is an internal KDM metamodel that allows the specification of software architectures along with the relationships among their elements [1], [2], [3]. It is important to note that the KDM metamodel currently has no easy way to instantiate it. The only alternative to doing this is an API called KDM-SDK that allows the use of Java Classes to create instances of the metaclasses and serialize it. Figure 1 shows some of these metaclasses.

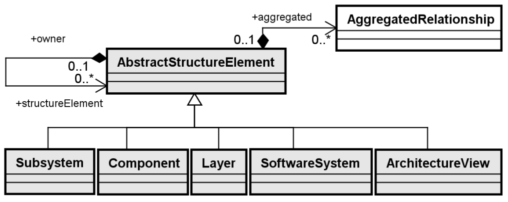


Figure 1. StructureModel class diagram. Adapted from [3]

The Structure Package contains the following five metaclasses for specifying architectural elements: `Layer`, `Component`, `Subsystem`, `SoftwareSystem` and `ArchitectureView`. By means of the self-relationship of the `AbstractStructureElement` metaclass, it is possible to create a composition among these elements. For example, it is possible to specify an architecture model composed of `x` subsystems, each subsystem including `y` layers and each layer including `z` components. Unlikely existing Architectural-Description Languages (ADLs), this package provides more specific architectural concepts, i.e., while the existing ADLs provide just modules, ports and connectors [18], this package provides terms that belong to architectural styles, like layers and components [16], [17].

This has an important impact, because architectural styles have access rules predetermined and well known. For example, it is well known that in strict layered systems, the above layer can just consume services provided by the layer below, and the layer below just can provide services to the layer immediately above [16]. Sarkar, Rama e Shubha [19] elaborate on two important constraints in the strict layering architectural style: the skip-calls and the back-calls. The skip-call happens when a layer has dependencies to other that is not the immediately below and the back-calls happens when a layer uses services from the layer above it.

In Figure 1 it is also possible to see the `AggregatedRelationship` (AR) metaclass. The goal of this metaclass is to represent relationships between architectural elements, like, between layers and components or between layers and subsystems.

Figure 2 graphically depicts an example of an AR instance between two layers. There are two elements (`l2` and `l1`), that are instances of the `Layer` metaclass. The relationship between these layers represents an instance of the `AggregatedRelationship` metaclass, where the source (from) of the relationship is the initial point of the arrows (`l2`) and the target (to) is the end point of the arrows (`l1`).

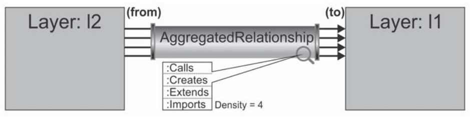


Figure 2. Example of a `AggregatedRelationship` instance

The AR act as a container, encapsulating primitive relationships. In KDM, primitive relationships are actions or structural dependencies, each one having its own metaclass. There are 34 metaclasses that represent these primitive relationships, like method calls (`Calls` metaclass), object creation (`Creates` metaclass), initialization value (`HasValue` metaclass), inheritance (`Extends` metaclass), macros (`Expands` metaclass) and data reading (`Reads` metaclass). Every `AggregatedRelationship` instance has a meta-attribute density, which represents the number of primitive relationships inside it. In Figure 2, the AR groups four primitive relationships.

3. The Approach

Our tool-supported approach is composed of two main parts: i) A DSL called DCL-KDM for specifying PAs in the ADM context; and ii) a Serializer for serializing PAs as KDM instance. DCL-KDM was built on top of an existing DSL called DCL [5]. We have evolved the original DCL in four main points.

The first one is the inclusion of keywords for specifying the architectural elements presented in Structure Package. We have included the following keywords: `layer`, `component`, `subSystem` and `softwareSystem`. The original version of DCL employs only the keyword `module`.

The second one was the possibility of specifying the level of the layers with the keyword `level`. Doing that, we were able to generate the access rules between the layers. Letting a software architect specifying a layered system, it is possible to predict and automatically generates the relationships between the layers, since strict layering is a well known architectural style [16], [17]. Therefore, the engineer does not need to write the access rules for that. This would not be possible just using modules.

The third one was the inclusion of the keyword `in` for describing composition of architectural elements. When an architectural element is said to be into another one, the element that is in the higher level can access everything of the internal one, but the opposite is not true. This feature also improves the productivity because it is also possible to automatically generate the access rules.

At last, the forth point is the serialization of the PA as a KDM instance. The original version employed a proprietary metamodel. This extensions consisted in the creation of a Serializer Module dedicated to generate a Structure Package instance representing the Planned Architecture. Our strategy for serializing the PA using the Structure Package is detailed in Section 3.3

3.1. Example of a PA

Figure 3 shows an hypothetical PA that is used along this paper. We are using this PA to represent architectural elements and also access rules among them. Every element is an instance of a KDM metaclass (format [`Metaclass : name`]). In this PA, there is a system `s1` represented by the larger rectangle. The `s1` is composed of two subsystems, `ss1` and `ss2`, represented by the smaller rectangles. Each subsystem is composed of components and layers. Inside some components there are layers and vice-versa.

Regarding the access rules, we are making them evident by using three graphical strategies. When we use the term "access", we mean any kind of primitive relationship. Table 1 (Section 3.2.2) shows some examples of these access in DCL-KDM. Our intention with these three graphical strategies is to represent: i) the access rules among layers (⟶); ii) the access rules among grouping of objects (▭); and iii) accesses rules among all the other architectural elements (⟹). At this point, it is important to note that there are only one-way arrows to represent dependencies. Thus, to

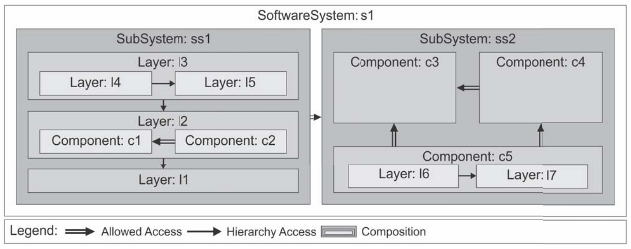


Figure 3. A Planned Architecture

represent interdependence of two elements it is necessary to use two simple arrows, one to represent in dependencies and another to represent out dependencies. We are adopting this notation to be more aligned with DCL-KDM concepts and to make the explanation easier along the paper.

The single arrow (⟶) is called Hierarchy Definition and it must only be used between layers. The goal is to make evident the hierarchy level between them and, as a consequence, their access rules [16], [17]. The source layer is in a higher level than the target layer. Therefore, the arrow can be read as "source Layer is higher than the target Layer", or "the source Layer uses the target Layer". For example, `l3` is in a higher hierarchy level than `l2`. So, it is possible to read "`l3` uses services provided by `l2`, but not the opposite".

The second strategy has the intention of representing composition (▭). The graphical representation is by grouping elements, i.e., one inside another one. In this notation the bigger element can access everything of the immediately internal one, but not the opposite.

The third strategy is the double arrow (⟹), which represents direct accesses among all the other architectural elements. As an example, we can read "component `c4` can access everything of component `c3`". The direction of the arrow is also important; sometimes an element may access another one but the opposite is not true.

In this notation the presence of arrows represent allowed accesses and the absence of them represent prohibited accesses. For example, there are no arrows between layers `l1` and `l3`, meaning the elements inside `l1` cannot access the elements of `l3` and vice-versa. The same applies to all the other elements that do not have arrows, for example, there is no arrow between the component `c4` and the layer `l1`.

Regarding the direction, an example is the relationship between layer `l6` and `l7`. There is a relationship between them, but it is unidirectional, so all code elements of `l6` can access the code elements of `l7`, but not the opposite. Regarding composition of elements, the component `c1` is a inner element of the layer `l2`, so the `l2` can access all elements inside the `c1`, but not the opposite.

Listing 1 depicts a DCL-KDM specification for the PA in Figure 3. The complete specification involves two blocks. The first block is called "`architeturalElements`" and second block is called "`restrictions`". In the first block the architect must specify the architectural elements, such as layers, components and subsystems and also their hierarchies and composition relations, which must be specified with the keywords “level” and “in...”. For example, in line 4 there is a declaration of a layer called l3, informing that its level is 3 and it is inside the subsystem ss1. It is important to mention that the bigger is the number of the level, the higher is the level of the layer.

```
1  architeturalElements {
2  (a) subSystem ss1;
3      subSystem ss2;
4  (a) layer l3, level 3, inSubSystem: ss1;
5  (a) layer l2, level 2, inSubSystem: ss1;
6  (a) layer l1, level 1, inSubSystem: ss1;
7      layer l4, level 2, inLayer: l3;
8      layer l5, level 1, inLayer: l3;
9      layer l6, level 2, inComponent: c5;
10     layer l7, level 1, inComponent: c5;
11     component c1, inLayer: l2;
12     component c2, inLayer: l2;
13     component c3, inSubSystem: ss2;
14     component c4, inSubSystem: ss2;
15     component c5, inSubSystem: ss2;
16 }restrictions {
17     c2 can-depend-only c1;
18     c4 can-depend-only c3;
19     c5 can-depend c4;
20     c5 can-depend c3;
21     ss1 can-depend ss2;
22 }
```

Listing 1. The PA Specification in DCL-KDM

In the second block (“restrictions”) the architect must specify the access rules of the architectural elements specified in the previous block. In this block the architect must specify more specific access rules. For instance, between lines 17-21 it is possible to see that there is one access rule between both subsystems (line 21). This access rule describes that the ss1 can access the ss2. That is, the elements in subsystem ss1 can have all kinds of access to the elements in subsystem ss2. Although some access rules can be automatically generated the architect can still decide to write some of them.

3.2. DCL-KDM Syntax

In this section we present the general syntax of the DSL. In Subsection 3.2.1 we explain three of the DCL-KDM keywords and in Subsection 3.2.2 we explain the keywords devoted for specifying access rules.

3.2.1. Architectural Elements Specification.

In this section we present three of the main keywords to describe architectural elements in DCL-KDM, witch are layer, component and subSystem.

Layers. The layers are represented through the keyword layer. In Listing 1 on lines 4-10 there are examples of the layer specification in DCL-KDM.

The expression for specifying a layer is divided into two mandatory parts and one optional. The first mandatory part is the use of the keyword layer, followed by its name and a comma, like “layer l3,” (line 4). The second one is the level of the layer by using the keyword level, followed by its value and a comma/semicolon, like “level 3,” (line 4).

The optional part is used when a layer is within other element, i.e., the architect should specify in which element the layer belongs to. The keyword used to this part depends on the contained element, in this case the container is a subsystem, so (s)he should use inSubSystem (line 4).

Component. The components are represented through the keyword component. In Listing 1 (lines 11-15) there are examples of the component specification.

The expression for specifying components is divided into one mandatory part and one optional. The mandatory part is the specification of the component through the keyword component, followed by its name and a comma/semicolon, like “component c1,” (line 11). The optional part is used when a component is within other architectural element, i.e., the architect should specify in which element the component belongs to. The keyword used to this part depends that the contained element, in this example,it is contained into a layer, so the keyword used is inLayer (line 11). Line 17 (Listing 1) depicts that unlike layers, the components requires that its access rules must be explicit specified.

Subsystems. The subsystems are represented through the keyword subSystem. In Listing 1 (lines 2-3) there are examples of the subsystem specification.

The expression for specifying a subsystem is divided into one mandatory part and one optional. The mandatory part is the specification of the subsystem through the keyword subSystem, followed by its name and a comma/semicolon, like “subSystem ss1;” (line 2). The optional part is used when a subsystem is within other subsystem, i.e., the architect should specify in which subsystem the subsystem belongs to. The keyword used to this part is the inSubSystem. As well as components, the subsystems constraints must be defined by the software engineer (line 21).

3.2.2. Access Rules Specification.

In this section we present the two ways to specify access rules in DCL-KDM; the manual and the automatic specification.

Manual Specification. In order to explain and define the access rules between architectural elements, we have opted for adopting the same terminology employed by Terra and Valente [5]. So, we have three distinguish forms to declare an access rule between two elements. Two of then can be seen in Listing 1 (lines 18-19), the third one was the opposite to the rule on line 18, being only c2 can-depend c1.

The expression for specifying an access rule is divided into three mandatory parts. There are two ways to specify the first mandatory part. The first one is the specification of the initial architectural element of the restriction through its name, like “c2” (line 17) The second one is if the engineer want to specify that just one element can access some other element. Thus the specification initiate with the keyword only followed by the name of the architectural element.

The second mandatory part is the specification of the access type of the access rule and its access, like “can-depend” (lines 19-21). In this part the DCL-KDM has three options for access type and ten options for access. The access type are can, cannot and must. The can represents that the architectural element can have the defined access with other specific element and it could be used with three different forms (can followed by the access, can followed by the access and only and can after the keyword only). The cannot represents that the architectural element cannot have the defined access with other specific element. At last, the `must` represents that the architectural element must have the defined access with other specific element. Already about of ten options for access, each one has its own mean. Table 1 depicts a short description of each access.

TABLE 1. ACCESS TYPES SUPPORTED IN DCL-KDM

Type	Short Description
access	Access of methods and attributes
declare	Declaration of variables
handle	Access and declaration of methods ans variables
create	Creation of objects
extend	Extension of classes/interfaces
implement	Implementation of classes/interfaces
derive	Extension and Implementation of classes/interfaces
throw	Throwing exceptions
useannotation	Use of annotations
depend	All previous

The lastly mandatory part is the specification of the final architectural element of the restriction through its name, followed by a semicolon, like "`c1;`" (line 17)

Automatic Generation. In DCL-KDM there are two access rules types that are automatically generated and do not need to be specified for architects: i) access rules for strict layering; and ii) access rules for object composition.

The access rules for strict layering are automatically generated thanks to the use of the keyword `level`. For example, the access rules in lines 1 and 2 shown in Listing 2 are automatically generated based on the lines 4, 5 and 6 of Listing 1. These two access rules are also responsible for the generation of other access rules to ensure the consistency of the PA. These other access rules are those in lines 3-6 (Listing 2) that deals with skip-calls and back-calls by means of generating one access rule between each two layers (like `i,j`) where the layer `i` cannot depend on the layer `j` when the level of the layer `j` is smaller than the level of the layer `i-1` or the level of the layer `j` is bigger that of the layer `i`.

```
1    l3 can-depend l2
2    l2 can-depend l1
3    l1 cannot-depend l2 {dealing with back calls}
4    l2 cannot-depend l3 {dealing with back calls}
5    l1 cannot-depend l3 {dealing with skip calls}
6    l3 cannot-depend l1 {dealing with skip calls}
```

Listing 2. Example of automatically generated access rules

The second access rules type automatically generated are the composition rules. In DCL-KDM this rules are automatically generated by means of the use the keywords `inSubsystem`, `inComponent` or `inLayer`. Through these keywords we are capable of verify which architectural elements is composed by others architectural elements. This is due to generate the composition between them and automatically generate the rules like in lines 1 and 2 of Listing 3, following the specification on lines 11 and 12 in Listing 1. These two access rules also implies that others access rules are generated to ensure the consistence of the PA. These other access rules are those in lines 3 and 4 in Listing 3.

However, it is important to highlight that engineers can still opt for not using the mechanism to automatically generated access rules and provide all the restrictions manually.

```
1    l2 can-depend c1
2    l2 can-depend c2
3    c1 cannot-depend l2
4    c2 cannot-depend l2
```

Listing 3. Example of automatically generated access rules

3.3. DCL-KDM Serialization

As the Structure Package was not originally designed for representing access rules of PAs, we had to decide how to represent the rules on it. Our decision was to represent the access rules by the presence/absence of `AggregatedRelationship` (AR) and also by the presence/absence of primitive relationship types inside the AR instance. Therefore the existence of an `AggregatedRelationship` instance between two elements implies they can communicate and the absence means they cannot communicate. At the same way, the presence of primitive relationships inside the AR means those relationships are allowed and the absence means they are prohibited. Depending on the access rule, there is an different set of relationships inside the AR container. In this way, the absence of the AR means all the 34 primitive relationship types are prohibited.

As an example, let us suppose Figure 2 is a PA. As there exist an `AggregatedRelationship` between the two layers it means these two architectural elements can communicate. This AR could be converted to the fallowing access rule "`l2 can-depend l1`". In this example, we had the source element (from - `l2`) and the target one (to - `l1`). However, to fully understand this communication, it is needed to analyze the primitive relationships inside the AR.

Inside the `AggregatedRelationship` instance there are four instances of relationships metaclasses of the KDM. Each instance represents a different type of relationship that is allowed between these elements. For instance, the `Calls` instance represents that it is allowed method calls from elements of `l2` to elements of `l1`. As the complete set of primitive relationships contains 34 elements, it is possible to know that there are 30 relationship types that are not allowed between these layers.

A KDM instance is serialized like a XMI file. In our case, to serialize a PA as a KDM instance, the file contains only instances of metaclasses of Structure Package and instances of metaclasses that represents relationships.

Listing 4 shows part of a PA serialized in XMI following the KDM format. Because of space limitations, we shown just the XMI that refers to the part highlighted with "(a)" in Listing 1.

This KDM instance have various elements that represents the Structure metaclasses. In this example, the elements shown are `subSystem`, `layer` and their access rules automatically generated. The metaclasses to representing the architectural elements can be seen on line 4 (`Subsystem` metaclass), lines 5, 12 and 19 (`Layer` metaclass). The metaclass to representing the access rules can be seen on lines 6 and 13 (`AggregatedRelationship` metaclass).

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <kdm:Segment xmi:version="2.0" [...] name="Planned
       Architecture">
3    <model xsi:type="structure:StructureModel" name="
         Planned Architecture">
4      <structureElement xsi:type="structure:Subsystem"
           name="ss1">
5        <structureElement xsi:type="structure:Layer" name
             ="l3"  [...]>
6          <aggregated from="//@model.0/@structureElement
               .0/@structureElement.0"
7              to="//@model.0/@structureElement.0/
                   @structureElement.1"
8              relation="//@model.1/@codeElement.0/
                   @codeElement.0/@actionRelation.0 [...]"
9              density="7"/>
10         [...]
11       </structureElement>
12       <structureElement xsi:type="structure:Layer" name
             ="l2" [...]>
13         <aggregated from="//@model.0/@structureElement
               .0/@structureElement.1"
14             to="//@model.0/@structureElement.0/
                   @structureElement.2"
15             relation="//@model.1/@codeElement.1/
                   @codeElement.0/@actionRelation.0 [...]"
16             density="7"/>
17         [...]
18       </structureElement>
19       <structureElement xsi:type="structure:Layer" name
             ="l1" [...]/>
20     </structureElement>
21   </model>
22   <model xsi:type="code:CodeModel"> [...] </model>
23 </kdm:Segment>
```

Listing 4. Part of a PA Serialized as a Structure Package Instance

The access rules are represented by means of the `AggregatedRelationship` metaclass. In this metaclass the direction of the access rule is represented by means of the `from` (lines 6 and 13) and `to` (lines 7 and 14) attributes. There is also an attribute called `density` that defines how many relationships the `AggregatedRelationship` contains and an attribute called `relation` that represents each relationship that the `density` evidentiate (lines 8 and 15). Like the attributes `from` and `to`, the attribute `relation` has the same quantity of XMI paths as the `density` attribute.

4. DCL-KDM Evaluation

In this section we describe two evaluations we have conducted. Below there are two GQM (Goal/Question/Metric) templates [20] for both evaluations.

First evaluation: **(i) object of study:** DCL-KDM; **(ii) goal/purpose:** comparing DCL-KDM with KDM-SDK to check which of them provides a better support in terms of productivity for software engineers; **(iii) perspective:** software engineers in need of specifying a PA in an ADM-based modernization project; **(iv) quality focus:** the easiness of specification of Planned Architectures comparing DCL-KDM and KDM-SDK in terms of lines of code and the quality of the generated instances **(v) context:** academic context. To guide this first evaluation we have developed two research questions: **RQ1:**"What are the pros and cons of using DCL-KDM and KDM-SDK for specifying PAs?"; and **RQ2:**"Does DCL-KDM contribute more to the correctness/quality of the generated instances than KDM-SDK?".

Second evaluation: **(i) object of study:** DCL-KDM; **(ii) goal/purpose:** comparing DCL-KDM with other three state-of-the-art techniques for specifying planned architectures. The analyzed techniques were: DCL [5], Dependency Matrix (DM) of the JArchitect tool [21] and Reflexion Models (RM) used by the SAVE tool [8]. **(iii) perspective:** software architects in charge of choosing a DSL for the specification of a Planned Architecture to any modernization project; **(iv) quality focus:** the specification of a PA, the use of the tool support and the scope of specification. **(v) context:** heterogeneous software systems/programs. To guide the second evaluation we have developed others two research questions: **RQ3:**"In terms of pros and cons, how is the use of the DCL-KDM comparing others state-of-the-art techniques?"; and **RQ4:**"The DCL-KDM can be used as a language to specify PAs as the others state-of-the-art techniques?".

4.1. Evaluation Strategy

4.1.1. First Evaluation. To conduct this evaluation we have used the hypothetical PA shown in Figure 3. We decided to use it because it involves a variety of architectural abstractions and combinations among them difficult to find in real systems. We claim that this PA is more suitable for exercising DCL-KDM than a real/big system with a fewer number of combinations.

This evaluation has involved two software architects from Academy who have high experience with ADM concepts, DCL-KDM and KDM-SDK. Their task was to specify three parts of the PA by using these both techniques. The first part was the layers `l1`, `l2`, and `l3` and their access rules. The second one was the components `c3`, `c4` and `c5` and their access rules and the third one was the components `c1` and `c2` inside layer `l2` and their relationships.

Listing 1 shows the whole specification of these three parts in DCL-KDM. Part 1 - Lines 4-6, Part 2 - Lines 13-15/18-20 and Part 3 - Lines 11-12/17. However, as the specification in KDM-SDK is much longer, Listing 5 shows just the specification of the first part (layers `l1`, `l2`, and `l3` and some access rules).

In Listing 5, it is possible to notice that the lines 2 and 3 instantiate a subsystem named `ss1`. In lines 4-9 the layers are declared (`l3`, `l2` and `l1`). In lines 10-12 the layers are added to the subsystem. In lines 13-16 a set of relationships are created, these relations represents the allowed access types between the architectural elements. Lastly, the lines 17-23 instantiate the `AggregatedRelationship` metaclass, define its owner, its direction and the relationships allowed between the two elements (`l3` and `l2`) by means of its attributes.

It is important emphasize that these two representations ("(a)"-Listing 1 and Listing 5) generate the same KDM instance. Listing 4 shows a snippet of the KDM instance (Structure Package metaclasses) serialized.

```
1 private void createFirstPartToEvaluated() {
2     Subsystem ss1 = StructureFactory.eINSTANCE.
          createSubsystem();
3     ss1.setName("ss1");
4     Layer l3 =StructureFactory.eINSTANCE.createLayer();
```

```
    l3.setName("l3");
    Layer l2 =StructureFactory.eINSTANCE.createLayer();
    l2.setName("l2");
    Layer l1 =StructureFactory.eINSTANCE.createLayer();
    l1.setName("l1");
    ss1.getStructureElement().add(l3);
    ss1.getStructureElement().add(l2);
    ss1.getStructureElement().add(l1);
    List<KDMRelationship> lisfOfRelationships = new
        ArrayList<KDMRelationship>();
    Calls relation = ActionFactory.eINSTANCE.
        createCalls();
    lisfOfRelationships.add(relation);
    [...]
    AggregatedRelationship newRelationship =
        CoreFactory.eINSTANCE.
        createAggregatedRelationship();
    newRelationship.setDensity(lisfOfRelationships.size
        ());
    newRelationship.setFrom(l3);
    newRelationship.setTo(l2);
    newRelationship.getRelation().addAll(
        lisfOfRelationships);
    l3.getAggregated().add(newRelationship);
    [...]
}
```

Listing 5. Part 1 of the PA (KDM-SDK)

4.1.2. Second Evaluation. In order to support the second evaluation and the discussion about the expressivity of the DCL-KDM, we conducted a comparison with the specification of a PA in DCL [5], DM [21] and RM [8].

The system chosen for this comparison was the myAppointments. This system was implemented by an software engineer of the Group of Software Engineering in PUC Minas and aims to manager personal information of agenda. Figure 4 shows the PA that the myAppointments was planned. The following constraints are the constraints that myAppointments should follow in its implementation [22]:

- Only view layer can depend on components of AWT/Swing.
- Only DAOs from the model layer can depend on database services. An exception is granted to the model. DB class, responsible for controlling database connections.
- The view layer can only depend on services provided by itself, by the controller layer, and by the util package (for example, to decouple data presentation from data access and view components can't access model components).
- Domain objects must not depend on the DAO, controller, and view types.
- DAO classes can only depend on domain objects, on other model classes allowed to use database services (such as model.DB), and on the util package.
- The util package must not depend on any class specific to the system source code.

Like the DCL-KDM is to specify PAs and in ACC process this specification is required, this system offers a strongly bases to our comparison, because it was implemented exclusively to evaluate ACC processes and tools [22].

4.2. Results

First Evaluation. This evaluation were conducted by focusing on the Lines of code (LoC) metric. This metric

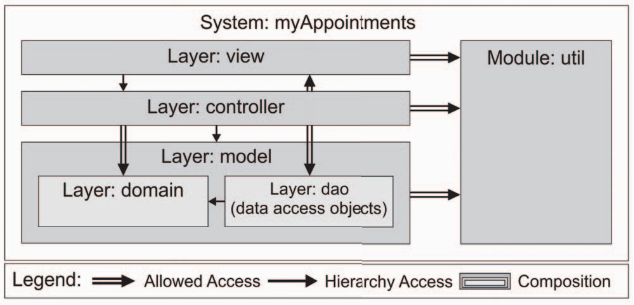


Figure 4. The Planned Architecture of myAppointments. Adapted of [22]

were chosen to provide an initial perception of the effort for creating the same specification in DCL-KDM and KDM-SDK. Table 2 shows the results for the specification of the three parts defined in Section 4.1.1 for each engineer.

TABLE 2. LoC FOR PA SPECIFICATION

Evaluated Part	Approach	LoC	
Part 1	DCL-KDM	4	4
	KDM-SDK	55	40
Part 2	DCL-KDM	6	6
	KDM-SDK	61	43
Part 3	DCL-KDM	4	4
	KDM-SDK	49	37

Second Evaluation. The specifications of the system myAppointments was performed by one software engineer with experience in these four state-of-the-art approaches (DCL-KDM, DCL, DM and RM). This evaluation were conducted by focusing in the use and in the pros/cons of each specification in each approach. As results, we had four specifications of the same system. Listing 6 shows the specification in DCL-KDM by using the Eclipse Plug-in. Listing 7 shows the specification in DCL by using the Eclipse Plug-in. Figure 5 shows the specification in DM by using JArchitect5 tool. And at last, Figure 6 shows the specification in RM by using SAVE tool.

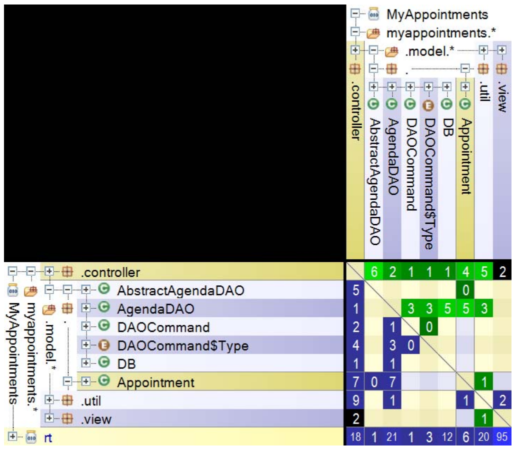
```
architeturalElements{
    layer view, level 3;
    layer controller, level 2;
    layer model, level 1;
    layer dao, level 2, inLayer: model;
    layer domain, level 1, inLayer: model;
    module util;
}restrictions{
    model can-depend util;
    view can-depend util;
    controller can-depend util;
    controller can-depend view;
}
```

Listing 6. myAppointment specification in DCL-KDM

```
%Modules
module Controller:    myapp.controller.*
module View:          myapp.view.*
module Model:         myapp.model.**
module Domain:        myapp.model.domain.*
module Util:          myapp.util.*
module DAO:           "myapp.model.[a-zA-Z0-9/.]*DAO"
module JavaAwtSwing:  java.awt.**, javax.swing.**
module JavaSql:       java.sql.**
```

```
%Constraints
only View can-depend JavaAwtSwing
only DAO can-depend JavaSql
View cannot-depend Model
Domain can-depend-only $java
DAO can-depend-only Domain, Util, javaSql
Util cannot-depend $system
```

Listing 7. myAppointments specification in DCL



	.controller	AbstractAgendaDAO	AgendaDAO	DAOCommand	DAOCommand$Type	DB	Appointment	.util	.view
.controller		6	2	1	1	1	4	5	2
AbstractAgendaDAO	5						0		
AgendaDAO	1			3	3	5	5	3	
DAOCommand	2		1		0				
DAOCommand$Type	4		3	0					
DB	1		1						
Appointment	7	0	7					1	
.util	9		1				1		2
.view	2							1	
rt	18	1	21	1	3	12	6	20	95

Figure 5. myAppointment specification in DM

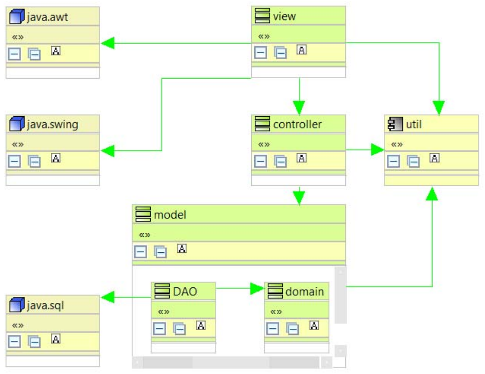


Figure 6. myAppointment specification in RM

4.3. Discussion of Results

In order to answer the **RQ1** and **RQ2** we focus in the results of the first evaluation, the LoC and our perception about the specification in DCL-KDM and KDM-SDK.

The RQ1 answer is organized here in pro and cons facts. A pro fact is that DCL-KDM automatically instantiates and serializes the PA as a KDM instance. Besides, it hides from the architect the details of how is the serialization, leaving (s)he free to concentrate on the specification of the PA and its concerns. It is also possible to specify a high amount of restrictions between the architectural elements, where some of then are automatically generated. Another pro fact is that to specify PAs in DLC-KDM it is not needed to have a deep knowledge of ADM and KDM metamodel. This occurs because the language is quite straightforward to be used and once it is learned, it is fairly easy to specify the PA.

A cons fact is that DCL-KDM just offers a fixed set of architectural elements. On the other hand, in KDM-SDK it is possible to define many architectural elements through inheriting a specific metaclass, as well as its composition.

In KDM-SDK there is no any restriction of composition, leaving out in the architect hands the decision of making any type of hierarchy or composition. That is, in DCL-KDM just some types of hierarchies and composition are possible. Another cons fact is that the in KDM-SDK there are no specific way to generate the specification, it is in the architect hands and its own knowledge. Already, in DCL-KDM it is required how the KDM will see the specification. That is, the same PA in KDM-SDK could be written in different forms, potentially increasing the possibility of errors or wrong specifications, but also increasing the possibilities of alternatives. In KDM-SDK there are no a way to representing restrictions. Leaving the engineer to specify your own style of restrictions between the elements. Such a fact that depending on the case may not be bad or wrong, while in DCL-KDM we force one specific way when we opted to use the presence/absence of primitive relationships into a `AggregatedRelationship` to represents the communication between the elements.

To answer the RQ2, the correctness is improved but the quality is the same. The correctness is improved mainly for two reasons: (i) the responsibility of defining constraints is largely removed from the software engineers, for instance, in layers the engineer must indicate at what level the layer belongs in order to automatically generate the constraints; and (ii) like is possible to see in the LoC values, each engineer has your own code style and logic, but using the DCL-KDM the specification and the logic to specify has to be the same, avoiding errors or mistakes in the specification. However, the quality remained the same for two reasons: (i) in DCL-KDM the generation of the KDM instance was automatically avoiding commons errors; and (ii) the two engineers that participate of the experiment has a deeply knowledge of the KDM generating a high quality KDM instance, an engineer that do not have the same knowledge possibly generates an instance with a worse quality.

In order to answer the **RQ3** and **RQ4** we use the results of the second evaluation focusing the use of the approach and how is the specification in each technique chosen.

To answer the RQ3, we separate a discussion about each approach. The specification in DCL-KDM it is simple and intuitively as the examples showed in the article, once the engineer understands how it is the syntax (s)he can specify an PA. Regarding the constraints of the myAppointment, the specification in DCL-KDM reaches five of the six constraints. While the DM reaches four of the six and the other two approaches reaches the six constraints. This is due to the deficit in DCL-KDM to specify specifics APIs (like `java.awt`, `java.sql`, etc.) and the DM to the rule's language be insufficient to express constraints based on the use of specifics interface types and name conventions.

Despite that, in comparing to the others techniques, the DCL-KDM is as comprehensive as the others, making it as an option to its use during the process to specify a PA.

The specification in DCL was similar to the specification in DCL-KDM. It was predictable once the DCL-KDM was based in the DCL. Despite that we observe quite interesting differences between them. One predictable difference is the possibility of specifying different architectural elements. Another one is that how each approach visualize the default access rules. In DCL, we observe that in the initial point of the specification all communication in the PA is allowed and the engineer needs to specify what is not allowed. Already, in DCL-KDM it is the opposite, all communication is not allowed and the engineer needs to specify what is allowed.

Other difference quite interesting is that the DCL needed that the software engineer has in your mind how the source code will be materialized. This is because in the specification of the PA it is necessary to initiate at least the package mapping between the source code and the PA. While in DCL-KDM and RM this mapping is not needed.

DMs are a compact and useful abstraction to visualize architectures, because they make it easier to engineer have a prompt zoom in and out over their system structure. But, the access rules language supported by the tools has revealed itself insufficient to express some restrictions. Another problem that we observe was the intrinsically need of the source code. In the JArchitect and others tools that we analyzed we observe that to have an DMs instance was needed to use the source code of the application. This leaves us to believe that is quite difficult to specify and use the DMs to generate a simple PA and its serializable version.

Out of the four techniques that we chose, RM and the SAVE tool is the only one that use supports a clearly defined ACC process by graphically means. The tool offers to an engineer to create a high-level model of the PA with all constraints needed. To use this tool support is ease and intuitive, generating high-level diagrams like in Figure 6. Despite that, the tool is proprietary one, consequently using a proprietary metamodel. So, it hurt the ADM principles using only ADM metamodels in the modernization processes making it difficult to promote the reusability. This happens because the solution act over one proprietary metamodel making it difficult and impossible the interchange among modernization tools of others authors.

Evaluating these four techniques, we observe two interesting facts that supports our answer to the RQ4. The first one is that the DCL-KDM is a quite acceptable technique to describe different PAs. The second one is that the basis DSL, could be translated in various others techniques just extending our KDM generation engine to generate others serializations types, like other metamodel, one instance of DM, etc. So, based on these facts the answer to RQ4 is yes, the DCL-KDM despite was generated for ADM context can be used as a language to specify PAs in any context.

5. Related Works

To the best of our knowledge, currently in the literature there is no a proposal focused on providing a contribution to the ADM context in terms of the PAs. All approaches found use proprietary metamodels and none of them mention the KDM metamodel as an alternative. In this way, the related works are those that present some support for specifying PAs; graphical or textual. There are a number of works that provide such support, but none of them focus on automatic generation of rules for strict layering and composition.

SAVE [8] is an ACC approach that identifies convergent, divergent and absent relationships in a system architecture. SAVE employs two proprietary metamodels: a high-level model for specifying the PA and a source code model for the current system implementation. LDM [9] relies on Dependency Structure Matrices (DSMs) to perform ACC. A DSM is a weighted square matrix whose rows and columns denote classes from an object-oriented system and the number of references from B to A is represented in the cell (A, B). Stafford and Wolf [15] have proposed a dependency analysis technique with ADLs. Their focus is on the representation of components, input and output ports and not architectural styles of higher level architectural components.

ReflexML [14] defines the traceability of UML component models to code using AOP type pattern expressions. Herold and Rausch [7] expresses 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.

ArchJava and ArchLint rely on AST as the underlying model for performing ACC. ArchJava [10] extends Java with architectural modeling constructs that seamlessly unify software architecture with implementation, ensuring that the implementation is according to the architectural constraints. ArchLint [11] is a data mining approach for ACC that identifies architectural violations based on a combination of static and historical source code analysis.

6. Conclusion

We have presented an approach for supporting software architects in the specification and serialization of PAs to be used in ADM-based projects. The primary usage of our approach is in Architecture-Conformance Checking (ACC) processes that occurs in ADM-based modernization projects.

An important aspect of our approach is showing it is possible to specify and serialize PAs with the original version of KDM and Structure Package. Therefore, we respect the standardization philosophy of OMG and propitiate a better level of reusability of tools that work with those standard metamodels. For example, modernization tools that already recognize the KDM standard, could easily take advantage of our approach reusing the DSL and the algorithm that generates the KDM instance as a PA.

Although in this paper we have highlighted the features that makes our approach more aligned with ADM-based projects, it can also be used in non-ADM projects with few adjustments. For example, the DCL-KDM can still be used only with the keyword "module" and by specifying all the access rules manually. However, the usage of the a

PA serialized as a KDM instance is a bit more difficult to be used in non-ADM projects. This happens because the algorithm that compares the Planned Architecture with the Current Architecture would have to deal with two different formats/metamodels. This makes the algorithms more complex and error-prone.

The Structure Package, in particular the `AggregatedRelationship`, has demonstrated to be powerful enough for our purposes. The possibility of representing relationships between architectural elements and encapsulate 34 types of primitive relationships inside it, has provide a good granularity level of representation.

We believe that DCL-KDM improves the productivity when specifying PA in ADM-based projects. This happens because architects do not have to write access rules for strict layering systems and composition of architectural elements. They just must inform the level of the layers (by the keyword `level`) and the composition of elements (by the keyword `in`). Based on these keywords the internal mechanism is be able to generate the access rules.

Furthermore, we conducted a preliminary evaluation divided in two parts. The first one to evaluated the DCL-KDM in the ADM context and the second one to show that despite the ADM context the DCL-KDM can be used as a language to specify PAs. The results show that DCL-KDM is an efficient alternative to KDM-SDK, which is currently the unique existing alternative for creating KDM instances (Structure Package). The results also show us that despite some differences between DCL-KDM and some state-of-the-art techniques the DCL-KDM can be used in different contexts to specify PAs in a high-level way.

As a future work we intend to improve our evaluation, by elaborating a controlled experiment with subjects. Our intention is to analyze it in a more deep way, each of the metaclasses and how DCL-KDM contribute for them. Our main focus is to evaluate its effectiveness to describe planned architectures in the ADM context, i.e., using the KDM as the main artifact.

Acknowledgments

We would like to thank the financial support of CNPQ, CAPES and Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), process number 2016/03104-0.

References

[1] OMG, "Object Management Group (OMG) Architecture-Driven Modernisation," *http://www.omgwiki.org/admtf/doku.php?id=start*, 2012.

[2] 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*, pp. 519–532, 2011.

[3] OMG, "Knowledge Discovery Meta-model (KDM)," 2016, available in http://www.omg.org/technology/kdm/, specification available in http://www.omg.org/spec/KDM/.

[4] L. De Silva and L. Balasubramaniam, "Controlling software architecture erosion: A survey," *J. Syst. Softw.*, vol. 85, pp. 132–151, Jan. 2012.

[5] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Softw. Pract. Exper.*, vol. 39, no. 12, pp. 1073–1094, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1002/spe.v39:12

[6] H. M. Sneed, "Estimating the costs of a reengineering project," in *12th Working Conference on Reverse Engineering (WCRE'05)*, Nov 2005, pp. 9 pp.–119.

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

[8] S. Duszynski, J. Knodel, and M. Lindvall, "Save: Software architecture visualization and evaluation," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, March 2009, pp. 323–324.

[9] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 167–176. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094824

[10] J. Aldrich, C. Chambers, and D. Notkin, "Archjava: connecting software architecture to implementation," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, May 2002, pp. 187–197.

[11] C. Maffort, M. T. Valente, R. Terra, M. Bigonha, N. Anquetil, and A. Hora, "Mining architectural violations from version history," *Empirical Software Engineering*, vol. 21, no. 3, pp. 854–895, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-014-9348-2

[12] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens, "Flexible architecture conformance assessment with conqat," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, May 2010, pp. 247–250.

[13] M. Abi-Antoun and A. Jonathan, *Tool support for the static extraction of sound hierarchical representations of runtime object graphs*. ACM, 2008, p. 743–744. [Online]. Available: http://www.cs.cmu.edu/ mabianto/papers/08-demo-extraction.pdf

[14] J. Adersberger and M. Philippsen, *ReflexML: UML-Based Architecture-to-Code Traceability and Consistency Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 344–359.

[15] A. L. W. Judith A. Stafford, "Architecture-level dependence analysis for software systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 4, pp. 431–452, Aug. 2001. [Online]. Available: http://dx.doi.org/10.1109/32.825767

[16] D. Garlan and M. Shaw, "An introduction to software architecture," Carnegie Mellon University Pittsburgh, Pittsburgh, PA, USA, Tech. Rep., 1994.

[17] B. Councill and G. T. Heineman, "Component-based software engineering," in *Component-based software engineering*, G. T. Heineman and W. T. Councill, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ch. Definition of a Software Component and Its Elements, pp. 5–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=379381.379438

[18] S. Hussain, "Investigating architecture description languages (adls) a systematic literature review," Ph.D. dissertation, 2013.

[19] S. Sarkar, G. Rama, and R. Shubha, "A method for detecting and measuring architectural layering violations in source code," in *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, Dec 2006, pp. 165–172.

[20] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[21] JArchitect, "Jarchitect 5: A tool to evaluate java code base," 2016, available in http://www.jarchitect.com/, Accessed on 12/2016.

[22] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. Mendonça, "Static architecture-conformance checking: An illustrative overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.