The Journal of Systems & Software 183 (2022) 111116

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Architectural conformance checking for KDM-represented systems☆

André de S. Landi [a], Daniel San Martín [b,d], Bruno M. Santos [b,*], Warteruzannan S. Cunha [b], Rafael S. Durelli [c], Valter V. Camargo [b]

[a] Amdocs digital network transformation communications, São Carlos, SP, Brazil
[b] Computing Department, Federal University of São Carlos - UFSCar, São Carlos, SP, Brazil
[c] Computing Department, Federal University of Lavras - UFLA, Lavras, MG, Brazil
[d] Exact Sciences Department, Universidad de los Lagos - ULagos, Osorno, Chile

ARTICLE INFO

Article history:
Received 8 April 2019
Received in revised form 8 July 2021
Accepted 4 October 2021
Available online 12 October 2021

Keywords:
Architecture-Driven Modernization
Knowledge Discovery Metamodel
Architecture-description language
Architectural-conformance checking
Current Architecture
Planned Architecture

ABSTRACT

Architecture-Driven Modernization (ADM) is a model-driven reengineering where systems are represented as instances of Knowledge Discovery Metamodel (KDM). KDM is the standard for representing systems in ADM context due to its power for capturing an extensive set of information about software systems. Besides, it is language and platform-independent, so every technique that is able of processing it also present this advantage. A recurrent activity in modernization projects is checking the conformance between the Current Architecture (CA) against the Planned Architecture (PA) in order to identify architectural drifts. The canonical phases of this activity are: (i) specification of the PA with its communication constraints; (ii) ex-traction of the CA, including the relationships among the architectural abstractions; and (iii) comparison between both architectures to identify the drifts. To the best of our knowledge, there is no ACC approach that addresses ACC in ADM context, considering KDM-represented systems. Therefore, we presents an ACC approach to be used in ADM context. We show how KDM can be used in ACC processes for representing the system to be modernized, the PA and the CA. We evaluated Arch-KDM using a small (LabSys-7KLoc) and a medium-size system (FreeMind-84KLoc) and the accuracy of the identification was acceptable.

1. Introduction

Architectural erosion is a well known and recurrent problem that affects the architecture of legacy systems. It is a gradual degradation of the architecture that leads to many problems, such as decreasing the reuse levels, breaking of the modularity, rising of the maintenance costs and generation of many undesired side effects along with the evolution of the system (de Silva and Balasubramaniam, 2012).

A technique that can be used to detect indications of architectural erosion is the Architecture-Conformance Checking (ACC), whose goal is to find the architectural drifts of the Current Architecture when compared to the architecture the system should have, i.e., its Planned Architecture (Knodel and Popescu, 2007; de Silva and Balasubramaniam, 2012; Terra and Valente, 2009). As soon as the drifts are detected, software architects can take more precise decisions about how to adjust the Current Architecture[1] towards the Planned one.

In order to conduct ACC processes, there are three moments/phases: the first one is the specification of the Planned Architecture, in that a representation of the intended architecture must be created; the second one is the recovering of the Current Architecture, in that a representation of the current implementation must be generated; and the third one is the comparison between both architecture representations in order to identify the drifts. A Planned Architecture is not a simple architectural specification since it must specify not only the required architectural abstractions (Layers, Components, etc.) the system must have but also the access/communication rules that must be guaranteed among these elements.

Architecture-Driven Modernization (ADM) was proposed for Object Management Group (OMG), and it is a way of conducting software reengineering employing models along the process. The central idea of ADM is to define and deliver standard metamodels to increase the success in modernization projects. Knowledge Discovery Metamodel (KDM) is the main metamodel of ADM as its

☆ Editor: W.K. Chan.

* Corresponding author.
E-mail addresses: andrelan@amdocs.com (A.d.S. Landi), daniel.santibanez@ufscar.br (D.S. Martín), bruno.marinho@ufscar.br (B.M. Santos), warteruzannan@estudante.ufscar.br (W.S. Cunha), rafael.durelli@ufla.br (R.S. Durelli), valtervcamargo@ufscar.br (V.V. Camargo).

[1] In this paper the authors assume that a Current Architecture as the same as the Implemented Architecture once the currently implemented code of the system will have its current architecture.

https://doi.org/10.1016/j.jss.2021.111116
0164-1212/

goal is to represent the system to be modernized. KDM is nowadays the *de-facto* standard for representing systems in the context of software modernization. What makes KDM distinguishable is that it is (i) an ISO (International Standard Organization) standard (ISO/IEC 19506); (ii) it is able of representing the complete spectrum of a software system, ranging from low-level details (such as source code and its actions) to higher-level abstractions, like architectural ones and business rules (Pérez-Castillo et al., 2011) and (iii) because it is language and platform-independent, what makes every algorithm or technique that manipulate it have this same advantage.

As a consequence of its broadness for representing the aspects of a software system, KDM is organized in packages that represent different abstractions of the system. Each package is also a metamodel responsible for representing one specific part of the system. The Structure package, for example, is the most crucial package in the context of this work. It contains metaclasses for representing the logical architecture of software systems (layers, components, and modules), the existing relationships between these architectural elements/abstractions, and also the implementation relationships between the architectural abstractions and the concrete source code elements (OMG, 2016).

Although there are several ACC approaches in the literature (Avgeriou and Guelfi, 2005; Ivkovic and Kontogiannis, 2006; Hannemann et al., 2005; Herold and Mair, 2014; Terra et al., 2012; Schröder and Riebisch, 2017; Koschke, 2018; Bandara and Perera, 2019), none of them investigates how to conduct ACC in ADM-based projects. What differentiates these approaches from ours is mainly the use of proprietary metamodels for representing the systems, which hinders the reusability of the algorithms that handle these metamodels. Up to this moment, there are little shreds of evidence of the applicability of KDM as a standard representation of systems in ACC processes.

This paper presents Arch-KDM, a tool-supported approach for supporting the conduction of ACC in ADM context. To do that, Arch-KDM uses KDM for representing the Current Architecture of the system to be modernized as for representing the Planned Architecture of the system. The Arch-KDM process involves four phases: In the first one, the Software Architect uses a DSL (Domain-Specific Language) called DCL-KDM for specifying the Planned Architecture (Landi et al., 2017). This Planned Architecture is then serialized as a KDM instance retaining all the architectural abstractions (Layers, Subsystems, etc.) and the allowed and prohibited access rules among them. In the second phase, the Software Architect uses a Wizard for recovering the Current Architecture of the legacy system that is also serialized as a KDM instance. In this phase, all the relationships between architectural abstractions and concrete source code elements are also recovered, providing a complete mapping between the abstractions and concrete elements. In the third phase, a comparison engine compares the Current Architecture with the Planned Architecture to search for differences, and in the last phase, the differences are presented as architectural drifts.

We have carried an evaluation focused on evaluating the Arch-KDM as a whole, analyzing the precision and recall of the drifts identification. The goal was to evaluate the precision, recall, and f-measure of the tool support provided by our approach.

The main contributions of this paper are: (i) Presenting an approach to conduct ACC for systems represented as KDM instances, therefore our approach acts over an ISO pattern and it is language-independent; (ii) Presenting evidences that KDM is a metamodel that can be used in ACC processes and (iii) KDM can be used for representing not only the system to be modernized but also the Planned Architecture. Such contributions make our approach platform and language independent, i.e., it can be used for checking the conformance of systems implemented in any language.

This paper is structured as follows: Section 2 introduces the concepts that are necessary to the foundation of this paper. Section 3 shows the Arch-KDM formalization, including an example of specification using the DCL-KDM, and the ACC steps supported by the tool. In Section 4 is presented the evaluation, the results, and the threats to validity. In Section 5 the related works and in Section 6 the conclusion are presented.

2. Background

2.1. Architectural-conformance checking

Architectural Conformance Checking (ACC) is one of the main activities in software quality control. ACC goals is to reveal the differences between the Planned Architecture and its real implementation (Knodel and Popescu, 2007). It reveals the relations and constraints foreseen by the PA that were violated by the system's implementation.

As a result of the ACC, it is possible to get three kinds of information. The first one is the relationships that were specified as "allowed" in the Planned Architecture that are implemented in the current system. These relationships are called "convergences", and they show that the implementation is compatible with the Planned Architecture. The second one is the relationships that were specified as "not allowed" in the Planned Architecture, but they are present in the current system. These relationships are called "divergence" or "architectural drifts", and they reveal that the implementation is not compatible with the Planned Architecture. The third one is the relationships that were not specified in the Planned Architecture, but they are present in the current system. These relationships are called "absences", and they show that the relations in the implementation were not found in the Planned Architecture.

There are two ways of conducting ACC processes (Knodel and Popescu, 2007; Murphy et al., 2001). The first one is the static verification in which the source code is compared with the Planned Architecture. The second one is the dynamic verification in which characteristics of the running system is compared with the Planned Architecture. Based on these two ways of conducting ACC process, the two main static techniques for performing ACC are Reflection Models and Compliance Relations Rules (Knodel and Popescu, 2007; Murphy et al., 2001). Reflection models is a technique that supports the use of a high-level system model as an eyeglass to see the source code model. Usually, this technique is applied when there is a few or none information about the system and its architecture (Murphy et al., 2001). The Compliance Relations Rules specify constraints between the architectural elements. These constraints can allow, prohibit, or impose the relations between the elements (Knodel and Popescu, 2007).

In our approach, we considered these two techniques. We used the Reflection Models to compare the architectural definitions and also applied the Compliance Relations Rules to verify if the architectural definitions are being respected. In our case, each relationship rule usually is composed of one relationship kind, a compliance rule kind, a source element, and a target element. In these rules, the source and target elements are defined by a regular expression that represents their names. The kind of compliance rule determines if the relationship between the components is allowed, prohibited, not defined, or imposed. Finally, the relationship kind defines which dependency kind exists between the elements, like a method call or an attribute declaration.

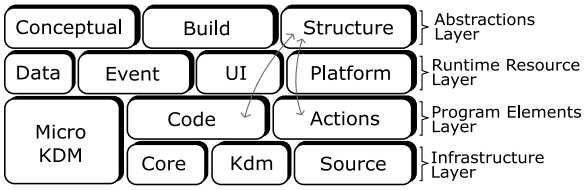


Fig. 1. The four KDM layers and packages from Pérez-Castillo et al. (2011).

2.2. Architecture-driven modernization and knowledge discovery metamodel

ADM is an OMG initiative to promote industry consensus on the modernization of the existing software system. It combines reengineering concepts, Model-Driven Architecture (MDA) principles and standard metamodels. ADM introduces several modernization standards like KDM, Abstract Syntax Tree Metamodel (ASTM) and Structured Metrics Metamodel (SMM) (Pérez-Castillo et al., 2011; OMG, 2017).

An ADM-based modernization process starts by reverse engineering a system into a KDM instance that, by its turn, is analyzed/mined to search for problems. Next, a set of refactorings and optimizations are performed to obtain a refactored and improved KDM instance (Durelli et al., 2017, 2014c). The process is completed with the generation of the modernized system. According to Perez (Pérez-Castillo et al., 2011), ADM can support many kinds of modernization scenarios such as: platform migration, language to language conversion, application improvement and architectural revitalization (Pérez-Castillo et al., 2011). KDM is the main metamodel of ADM and able to represent all the characteristics of software systems in a unique metamodel (Pérez-Castillo et al., 2011). A schematic representation of KDM can be seen in Fig. 1. It is divided into four layers that are further divided into packages. Each package is an internal metamodel, concentrating on specific aspects of the software. Thus, there are packages for representing a wide spectrum of systems abstractions, from low-level details like source-code (`Code` package) and run-time actions (`Action` package) to high-level details like User Interface (`UI` package), Business Rules (`Conceptual` package) and Architectural View (`Structure` package) (OMG, 2016, 2017).

It is important to mention that although KDM is divided into layers, its packages can communicate with each other. These inter-package communications are a key point in KDM since it allows mapping higher-level abstractions to lower-level ones. These communications between the packages are schematically represented in Fig. 1 by the two arrows from `Structure` to `Code` and `Action` packages. These three packages are the most important packages in the context of our research.

Code Package contains all the metaclasses for modeling the source code static structure. For example, `ClassUnit` metaclass represents classes and `InterfaceUnit` metaclass represents interfaces. The Code package has a total of 90 metaclasses and all the abstract elements for representing the source code (OMG, 2016). The Action Package defines metaclasses to represent behavioral units. Examples of these behaviors are: declarations (`Reads`, `Creates`, etc.), operators (`Writes`, `Addresses`, etc.), and flow conditions (`Flow`, `TrueFlow`, etc.). When generating a KDM instance, it is assumed that each element of the Action package corresponds to a behavior in a programming language.

The Structure Package is one of the most important ones as it is able to represent the logical architecture of a software system. Fig. 2 shows in the left part, the Structure package metaclasses in gray and other important related metaclasses in white. In the right part, there is a schematic representation of a Structure package instance. Structure package provides five metaclasses for representing architectural elements: `Subsystem`, `Component`, `SoftwareSystem`, `ArchitectureView` and `Layer`. Besides, by means of the self-relationship of the `AbstractStructureElement`, it is possible to create a hierarchy among these elements. For instance, it is possible to create an architecture having a Software System with two subsystems, which include two layers each, where each layer can include two components.

This package also provides an important means for specifying mappings between higher-level concepts to lower-level ones. This can be seen like an abstract-concrete mapping and this is done by an attribute named "`implementation`" (OMG, 2016), represented by the relationship between the AbstractStructureElement and KDMEntity metaclasses. Notice that KDMEntity metaclass belongs to the Core package, which is a central KDM package that provides base metaclasses for the other packages. `KDMEntity` is one of the most important metaclasses, since all the other KDM metaclasses are direct or indirect subclasses of it. Thus, all KDM metaclasses are KDM Entities.

`AggregatedRelationship` is another important metaclass herein because its role is to capture relationships among architectural abstractions. It is a kind of relationship that can group other primitive relationships within it. This is being represented in the Figure by the 0..n association between the `AggregatedRelationship` metaclass and the `KDMRelationship` metaclass. In KDM, every relationship type is represented by a metaclass, examples of primitive relationships are method calls (`Calls` metaclass), object instantiation (`Creates` metaclass) and implements relationships (`Implements` metaclass). Each `AggregatedRelationship` involves two KDM Entities, the source (*from* property) and target (*to* property), as can be seen in part A of Fig. 2.

Since all the architectural elements are KDM Entities (due to the inheritance), it is possible to represent relationships between these architectural elements employing the `AggregatedRelationship`, which is schematically shown in Fig. 2 Part **B**. In the example, we have a relationship between the layers Controller and Model. The cylinder between the two layers represents an instance of the `AggregatedRelationship` metaclass. The controller layer represents the source (*from*) of the relationship, and the model layer represents the target (*to*) of the relationship. An aggregated relationship incorporates primitive relationships inside itself. Primitive relationships are "actions" or structural dependencies that are also represented as KDM metaclasses. In Fig. 2 Part **B**, they are represented by the set of arrows that connects the two layers through the `AggregatedRelationship` instance. Every `AggregatedRelationship` has a density, which represents the number of primitive relationships inside it. In this example, the density is six since it involves six relationship instances (*calls*, *extends*, *creates*, *reads*, *imports*, and *hasType*).

An important point here is regarding the types of relationships presented in KDM. Some of them have canonical names that makes easy to understand what they really are in source code, such as: calls, extends, imports, etc. However, there are some other terms that ask for an additional explanation. The terms are:

- HasType. This type of relationship occurs when a source code element has the type of another source code element;
- UsesType. This type of relationship occurs when there is a line of code that makes a data conversion.

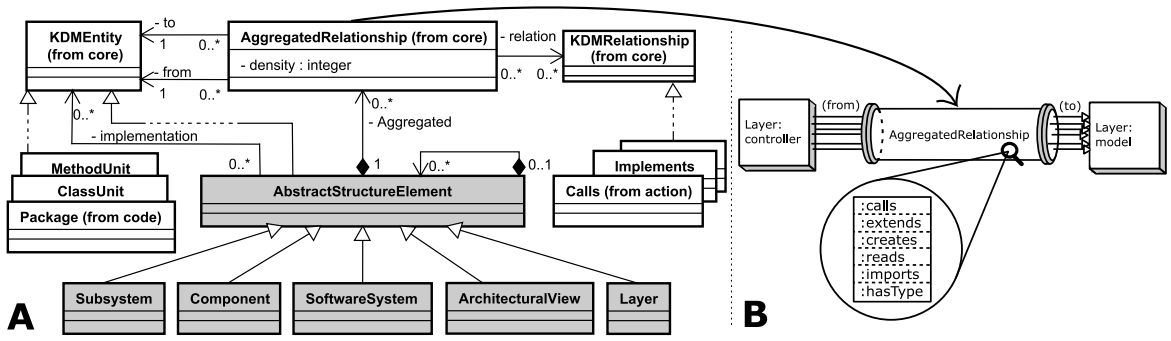


Fig. 2. A - Structure Package Class Diagram from OMG (2012); B - Schematic example of a Structure Package Instance.

3. Architectural conformance checking with the Arch-KDM approach

The Arch-KDM approach is divided into four phases, as can be seen in Fig. 3. The first phase is denominated as "Planned Architecture Specification", and to support this step, we provide a DSL denominated DCL-KDM (Landi et al., 2017; Chagas et al., 2016) with this DSL, the software engineer can generate a file with an extension named "`.dcl`". This file is serialized by means of the "`PASerializer-KDM`" generating the Planned Architecture in KDM format (**A** artifact). The second phase is the "Current Architecture Extraction", where the software architect must interact with a Wizard to recover the current architecture of the system. This process also uses the **A** artifact to map the current source code properly. The output of this step is another KDM instance containing the current architecture of the system (**B** artifact).

The next two steps are automatics; the third phase is named as "Architectures Comparison", this is the primary step of our approach. It is in this step that we get the two artifacts generated in the previous steps (**A** artifact and **B** artifact) and performed an algorithm that accomplishes the ACC and discover, based on the KDM instances, the system violations. The final phase is denominated as "Architectural Drifts Visualization", in this step the violations (**C** artifact) are grouped by means of the "`DriftDicovery-KDM`" to turn these violations in architectural drifts. It is important to highlight that our approach is a generic one, that is, we elaborated and implemented with the conventional steps of any ACC approach (Terra and Valente, 2009; Terra et al., 2012; Maffort et al., 2016; Knodel and Popescu, 2007; Murphy et al., 2001). All developed algorithms rely exclusively on KDM metamodel. Therefore, in theory, our approach can works efficiently for checking the conformance of any system implemented in an object-oriented language. In the next sections, we detail each of the four presented phases.

3.1. Planned architecture specification

In this phase, the goal is to create a Planned Architecture using the principles of ADM. To achieve this goal, it is necessary to create a specification of the architectural elements and their restrictions. In this way, we created the DCL-KDM, which is composed of two main parts: (i) A Domain-Specific Language (DSL) for specifying the Planned Architecture in the ADM context, and (ii) the PASerializer-KDM (Fig. 3) for serializing the Planned Architecture as KDM instance.

The DCL-KDM is an extension of the DSL proposed by Terra and Valente (Terra and Valente, 2009), so to turn the DCL into DCL-KDM we performed three main extensions - (*i*) change the DCL's grammar to add new keywords for representing KDM's metaclasses; (*ii*) automate the generation of constraints; and (*iii*) perform the serialization of the Planned Architecture in a KDM instance. More information regarding the syntax and details of each extension of the DCL-KDM can be seen in our previous work named "Supporting the Specification and Serialization of Planned Architectures in Architecture-Driven Modernization Context" (Landi et al., 2017).

The basis of the specification in DCL-KDM is modularized in two main blocks, "`architeturalElements`" and "`restrictions`". Listing 1 depicts an example of DCL-KDM specification for a Planned Architecture.

The first block (lines 1–12) is where engineers specify the architectural elements of the Planned Architecture. This specification also has their hierarchies and composition relationships specified with some keywords, like "`level`" and "`inLayer`". For instance, in line 4, there is a declaration of a layer called `view`, informing that its level is 3, and it is inside the subsystem `core`. It is important to mention that how much bigger the level value, the layer has a higher level in its hierarchy. The second block (lines 13–20) is where engineers specify the constraints between the architectural elements specified. This block must contain the constraints between the elements. For instance, it is possible to realize that there are two constraints between `controller` and `repository` (lines 14 and 15). These constraints describe that the `controller` can communicate with `repository` and vice-versa. It is important to mention that in this case, when we said that they can communicate, is that they can have the primitive relationships explained in Section 2.

```
1  architecturalElements{
2      subSystem core;
3
4      layer view, level 3, inSubSystem: core;
5      layer controller, level 2, inSubSystem: core;
6      layer model, level 1, inSubSystem: core;
7
8      component repository, inLayer: model;
9      component converter;
10     component generic;
11
12     module validator;
13 }restrictions{
14     controller can-depend repository;
15     repository can-depend controller;
16
17     only controller can-depend validator;
18     only controller can-depend converter;
19     only controller can-depend generic;
20 }
```

Listing 1: An example of Planned Architecture with DCL-KDM

Once the architectural specification was completed, like Listing 1, it is possible to serialize the Planned Architecture in terms of KDM, by the PASerializer-KDM (Fig. 3). This serialization is guided by the KDM, and it is materialized in an XML file that

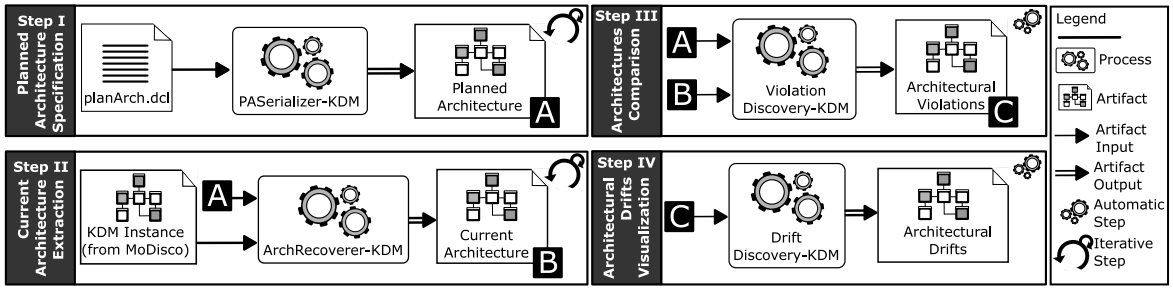


Fig. 3. Arch-KDM approach.

represents the architectural elements and its relationships as an instance of the KDM (Landi et al., 2017). For the first block, the serialization is a simple map by each architectural element and its representation in KDM, but for the second block, the KDM does not give us a clear support for constraints between the architectural elements.

Therefore, regarding to the second block, we decided to represent constraints using the metaclass `AggregatedRelationship`. This metaclass is like a container to primitive relationships, as illustrated in Section 2. To use this `AggregatedRelationship`, we create an instance for representing the communication between two architectural elements. Then, the presence of an `AggregatedRelationship` indicates that there is communication between these elements, and the absence indicates that there is none communication. Also, to take advantage of KDM we deepen our decision by using the 34 primitive relationships that KDM provides to us. These relationships work in a similar way that the Aggregated. The presence/absence of primitive relationship types inside the `AggregatedRelationship` instance state that these types of relationships can exist between the architectural elements.

For instance, the absence of the `AggregatedRelationship` between two architectural elements A and B represents that there are none of the 34 primitive relationships between them. In the opposite side, the presence of the `AggregatedRelationship` between two architectural element A and B represents that there is a specific set of primitive relationships between them, so, if there is only the relationship type "method calls (`Calls` metaclass)" inside it, then it represents that between A and B only this type is allowed.

3.2. Current architecture extraction

It is important to state that a PA is not an artifact that does not need maintenance. Changes in the organization culture, market demands and other internal and external factors can trigger updates in the PA. Therefore, software architects need to be aware of that and establish a systematic procedure for monitoring and checking whether the PA keeps meeting the desired goals.

Therefore, the goal here is to obtain a representation of the Current Architecture of the system so that it can be compared to the Planned Architecture obtained in the previous phase. Our strategy is to materialize the Current Architecture as KDM instances so that it can be more easily compared, in an automatic way, to the Planned Architecture, which is also materialized as a KDM instance.

In our approach, Current Architecture is an artifact that owns these four items:

(i) ***All the concrete source code elements and their relationships.*** These source code elements include packages, classes, methods, attributes, method bodies, parameters, and also relationships, such as calls, implementations, inheritances, parameter passing. All source code elements are represented by the Code and Action Packages of KDM;

(ii) ***All the architectural abstractions of the system.*** Architectural abstractions include layers, modules, subsystems, and components. The Structure Package of KDM represents architectural abstractions;

(iii) ***All the relationships between architectural abstractions.*** This kind of relationships says, for example, that Layer A communicates with Layer B. This kind of relationship is represented in KDM through the `AggregatedRelationship`, and it resembles UML associations;

(iv) ***The concrete-abstract mappings***. These are the mappings between the architectural and source-code levels, linking abstractions to concrete source code elements. This kind of mapping says, for example, the Layer A is represented in the source code by the package P. This is represented in KDM by the *implements* relationship.

The process of extracting the Current Architecture is performed in two steps: ***(1)* extracting the source-code view of the system** by reverse engineering it using *MoDisco* (Bruneliere et al., 2010; Brunelière et al., 2014) and; ***(2)* reconstructing the architectural view of the system**; which is done semi-automatically. In the first step *MoDisco* (Bruneliere et al., 2010; Brunelière et al., 2014) generates viewpoints of the system related to source code and its relationships, that is, the Code and Action Packages of KDM. As MoDisco is unable to reconstruct the architectural viewpoint of the system, the second step is responsible for complementing this lack by generating the KDM Structure Package.

The second step, which is the **reconstruction of architectural view**, is performed in two sub-steps called: **Mapping Abstractions** and **Reconstructing Relationships between Abstractions**. The first one is manual, and the second is automatic. The first one must be conducted by a software architect that must map the architectural abstractions, declared in the Planned Architecture, to the concrete source code elements of the system. To be able to perform this activity, the architect must know the high-level architecture of the system, or he must have access to the architecture documentation if any. For example, to be able to map Layer A to Package A, (s)he must previously know that this package is the source code representation of that layer. Therefore, our approach helps in the identification of the fine-grained drifts, i.e., mostly internal relationships among packages. If the architect is not able to create the mappings, the conformance checking cannot be conducted.

Fig. 4 shows a screenshot of the ArchKDM Wizard, where software architects can work by mapping the architectural abstractions of the system to its source-code elements. Letter **A** points out a window in which appears abstractions that were declared in the Planned Architecture. It is possible to see the existence of a subsystem with three internal layers; Layers A, B, and C. Letter **B** points out a window where the Code Elements of the system are shown. Letter **C** is the windows in which appears

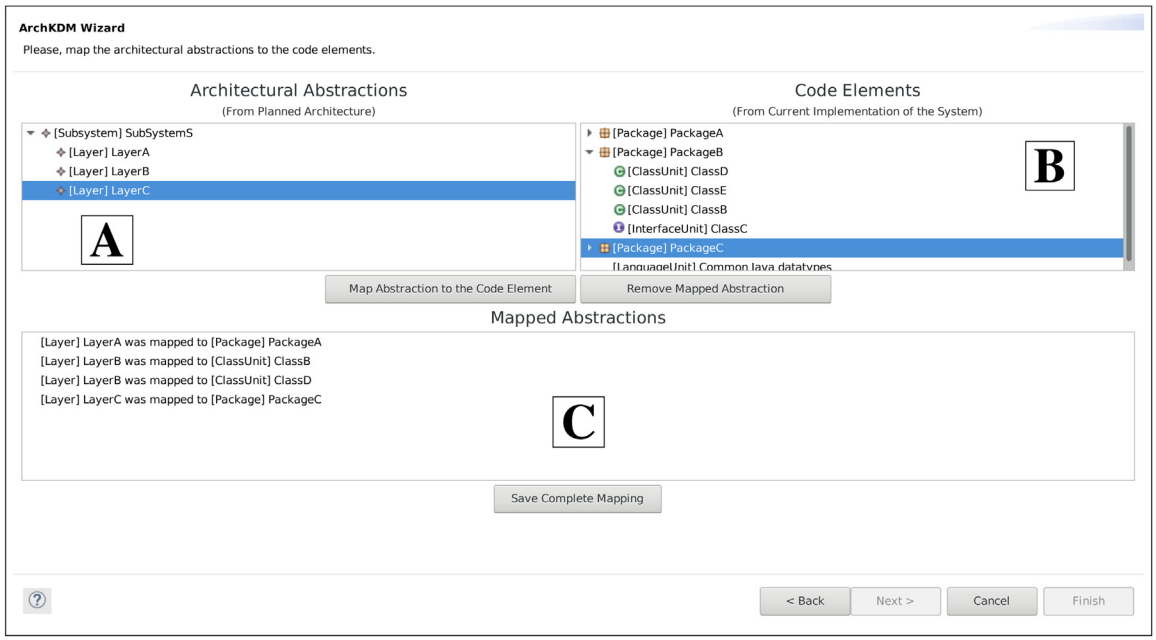


Fig. 4. Mapping abstractions with code elements.

the mappings between architectural abstractions and source code elements already done by the Software Architect. Formally, let $(X, \{a_1, a_2, \ldots, a_k\})$ be the relation between elements of the set of abstractions and the subset of code elements, that means for each abstraction X there is $k \geqslant 1$ such that X is mapped to $\{a_1, a_2, \ldots, a_k\}$. For instance, Fig. 4 presents 3 mappings:

$(LayerA, \{PackageA\})$; $(LayerB, \{ClassB, ClassD\})$;
$and(LayerC, \{PackageC\})$

Our Wizard also allows mapping architectural abstractions not only to container elements, such as packages but also to classes and interfaces. Our example shows that Layer A and C are mapped to packages, which is the most conventional mapping. However, we also show that Layers can be mapped to classes; this is the case of Layer B, which is represented in the source code for classes B and D.

As soon as the Software Architect does the mapping process, the second automatic sub-step (Reconstructing Relationships between Abstractions) can be started, which aims at identifying and reconstructing the relationships among the Architectural Abstractions. This process involves the execution of three algorithms that generate the instances of `AggregatedRelationship` metaclass and its primitive internal relationships.

This is a bottom-up process because of discovering relationships between architectural abstractions. Firstly it is necessary to identify relationships between source-code elements. For instance, suppose *ClassA* owns a Call relationship with *ClassB* and they belong to different packages; *Package A* and *Package B*. Suppose also *Package A* is the concrete materialization of the architectural abstraction *LayerA*, i.e., *Package A* was mapped to *LayerA* and *Package B* to *LayerB*. Therefore, it is clear that there is a relationship between *Package A* and *Package B*. As they are mapped to Layer A and B, we can say there is also a relationship between *Layer A* and *Layer B*. Since relationships between architectural abstractions are represented with the `AggregatedRelationship` in KDM, in this case, we create an instance of a `AggregatedRelationship` between *Layer A* and *Layer B* (if there is none) and insert an instance of the Call primitive relationship inside it. As new primitive relationships are discovered between Package A and B, they are also inserted into the already existing `AggregatedRelationship` that links Layer A and B.

Next, we explain the three algorithms involved on the reconstruction of the relationships between architectural abstractions. Algorithm 1 is the main one and uses the other two (Algorithms 2 and 3) shown below. This algorithm initiates creating a copy of the Current Architecture and the Planned Architecture into a new and unique KDM instance, by the method `copyInformation()`. After that, the "`for`" loop is responsible for analyzing the concrete-abstract mappings (provided by the Software Architect) and generating relationships of the type "`implements`", which is used in KDM for linking abstractions to concrete source code elements. The method "`getCodeInfo()`" retrieve this information based on the "`concreteAbstractMappings`", "`architectureAbstraction`" and "`code- Information`", this method is not presented in this paper. Finishing this first mapping, we initiate the discovery of the `AggregatedRelationship`.

Algorithm 2 shows the core of the discovering of `AggregatedRelationship` between the architectural abstractions. This algorithm receives as an input the KDM instance with the information of the Current Architecture, Planned Architecture, and the initial mapping between them. The process starts with the iteration of each architectural abstraction already mapped by the implementation property of each architectural abstraction. On each iteration, the algorithm gets the code elements that represent the architectural abstraction and initiate another iteration. This second iteration gets the code element that represents the implementation of the architectural element and recover (method "`recoverPrimitiveRelationships`") the existing relationships by each type of the 34 types of KDM relationships. When it finds the existence of a group of relationships between code elements that belong to different abstractions, the algorithm creates or updates (method "`createOrUpdateAggregateds`") an instance of the `AggregatedRelationship` KDM metaclass to make explicit the relationship between the architectural abstractions.

The method "`recoverPrimitiveRelationships`" (not presented in this paper) searches the primitive relationships of each architectural abstraction based on the primitive relationship type

Algorithm 1: GENERATING THE CURRENT ARCHITECTURE

```
Input: codeInformation: KDM instance containing the code of the system; plannedArchitecture: KDM instance containing the
       Planned Architecture; concreteAbstractMappings: The concrete-abstract mappings provided by the Software Architect in
       the Wizard.
Output: currentArchitecture: KDM instance containing the Current Architecture fully mapped
1  begin
2      copyInformation(currentArchitecture, codeInformationKDM)
3      copyInformation(currentArchitecture, plannedArchitecture)
4      for each architecturalAbstraction ∈ plannedArchitecture do
5          currentArchitecture.plannedArchitecture.get(architecturalAbstraction)
6          .implementation(getCodeInfo(concreteAbstractMappings, architecturalAbstraction, codeInformation)
7      end
8      discoverAggregatedRelationships(currentArchitecture)
9  end
10 return currentArchitecture
```

Algorithm 2: METHOD DISCOVERAGGREGATEDRELATIONSHIPS

```
Input: currentArchitecture: KDM instance containing the Code Information of the Current Architecture and the Structure
       information of the Planned Architecture
Output: Update of the currentArchitecture by reference
1  begin
2      for each architecturalAbstraction ∈ currentArchitecture do
3          for each codeElement ∈ architecturalAbstraction do
4              for each relationshipType ∈ RelationshipTypes do
5                  relationshipsOfCodeByArchitecturalAbstraction ←
                   recoverPrimitiveRelationships(relationshipType, codeElement, currentArchitecture)
6                  createOrUpdateAggregateds(architecturalAbstraction,
7                  relationshipsOfCodeByArchitecturalAbstraction)
8              end
9          end
10     end
11 end
```

received by parameter. It uses an SDK named KDM-MANAGER[2] responsible for performing searches in KDM instances and retrieve information about it. The SDK is used to recover all types of relationships that are inside the code elements of an instance. After finding them, the set of relationships are separated by the target architectural element to simplify future algorithms.

At last, the Algorithm 3 shows the method "`createOrUpdateAggregateds`". This method has the logic of how an instance of `AggregatedRelationship` KDM metaclass is created or updated. In this method it is performed an iteration on each target architectural element to search for existing aggregated with the "`from`" architectural element and the "`to`" architectural element from all the possible options of the relationships founded in the method "`recoverPrimitiveRelationships`". In negative case, the aggregated with that "`from`" and "`to`" did not exists, so we create a new one between these elements. Then, the instance (new or not) is updated adding the relationships founded and updating the density of the `AggregatedRelationship`.

This process of using the methods "`recoverPrimitiveRelationships`" and "`createOrUpdate Aggregateds`" is repeated for each one of the 34 relationships types that KDM provide to us for each architectural abstraction selected by the software architecture. As the main result of this phase, we have a KDM instance with the Structure and Code Packages linked. That fully mapped KDM instance enables us to perform the conformance checking explained in the next section.

[2] This SDK was created by the primary author to standardized and facilitate the handling with KDM instances in Java. This SDK can be seen and cloned on the https://github.com/dedeLandi/kdm-manager repository.

3.3. Architectures comparison - detecting violations

In this phase, the goal is to automatically compare the two artifacts created (Planned Architecture and Current Architecture) for identifying the violations between these both architectural representations. To achieve this goal, we create an algorithm that analyzes these two KDM specifications and identify the fine-grained violations.

The Algorithms 4, 5 depicts the developed algorithm that uses the Planned Architecture to check the relationships of the Current Architecture to evaluate if the relationship can exist or if the relationship is a violation. This algorithm contains two main parts. The first one is an iteration between the relationship allowed by the Planned Architecture as in the relationships that exist in the Current Architecture, and the second one is the processing of the instance to verify if it is a violation.

The Algorithm 4 represents the core of the ACC algorithm. This algorithm initiates recovering the instances of `AggregatedRelationship` in both inputs. Then, it is performed an iteration in each one of the planned `AggregatedRelationship` instances. Therefore, some data of the instance of the iteration is recovered, and it is searched in the Current Architecture the instance that is parallel to it. Completing this processing, we had an instance of the Planned Architecture and a set of instances of the Current Architecture. Then initiate another iteration in the set of the current `AggregatedRelationship` instances to validate if the relationships inside it are violations through the method `removePrimitiveRelationship`.

Algorithm 3: Method createOrUpdateAggregateds

```
Input: relationshipsOfCodeByArchitecturalAbstraction: Relationships to analyse; architecturalAbstraction: The architectural
       abstraction to analysed
1  begin
2      for each architecturalAbstractionTo ∈ relationshipsOfCodeByArchitecturalAbstraction.allArchitecturalAbstractions do
3          if architecturalAbstractionTo ∉ architecturalAbstraction.allAggregatedRelationships.to then
4              architecturalAbstraction.createAggregatedRelationshipWithTo(architecturalAbstractionTo)
5          end
6          architecturalAbstraction.getAggregatedRelationshipWithTo(architecturalAbstractionTo).add(
7          relationshipsOfCodeByArchitecturalAbstraction.getRelationships())
8          architecturalAbstraction.getAggregatedRelationshipWithTo(architecturalAbstractionTo)
9          .updateDensity()
10     end
11 end
```

Algorithm 4: Applying the Architectural Conformance Checking

```
Input: plannedArchitecture: KDM Instance of the Planned Architecture; currentArchitecture: KDM Instance of the Current
       Architecture
Output: currentArchitecture: KDM Instance of the Current Architecture with a new model composed of violations
1  begin
2      possibleAggregateds ← getAggregatedsFrom(plannedArchitecture)
3      actualAggregateds ← getAggregatedsFrom(currentArchitecture)
4      for each allowedAggregated ∈ possibleAggregateds do
5          from ← allowedAggregated.getFrom
6          to ← allowedAggregated.getTo
7          allowedRelationships ← allowedAggregated.getRelationships
8          currentAggregateds ← recoverSameAggregated(to, from, actualAggregateds)
9          for each currentAggregated ∈ currentAggregateds do
10             removePrimitiveRelationship(allowedRelationships, currentAggregated)
11         end
12     end
13 end
14 return currentArchitecture
```

Algorithm 5: Method removePrimitiveRelationship

```
Input: allowedRelationships: Relationships allowed by
       Planned Architecture; currentAggregated:
       AggregatedRelationship instance of the Current
       Architecture to analyse
1  begin
2      for each relationshipToRemove ∈ allowedRelationships do
3          for each currentRelationship ∈
           currentAggregated.getRelationships do
4              if relationshipToRemove.class =
               currentRelationship.class then
5                  currentAggregated.remove(currentRelationship)
6                  currentAggregated.updateDensity()
7              end
8          end
9      end
10 end
```

The Algorithm 5 represents the method `removePrimitiveRelationship`. This method is responsible for removing the relationships that are not violations. The method receives as inputs a set of relationships allowed by the Planned Architecture and an `AggregatedRelationship` instance of the Current Architecture. Then it is performed an iteration in each relationship allowed. In each iteration is performed another iteration in the relationships inside the `AggregatedRelationship` instance. So, it is validated if the relationship inside the `AggregatedRelationship` has the same type of relationship allowed. In the positive case, the relationship inside the `AggregatedRelationship` is removed from it. In the negative case, the relationship is not removed. In this way, after finished the algorithm, the relationship presented inside all `AggregatedRelationship` instances of the Current Architecture are violations.

3.4. Architectural drifts visualization

This phase's goal is presenting to the software architect all violations found by the tool in terms of architectural drifts. The result of the third phase is a KDM instance containing the violations, each of these represents a small part of the architectural problem, so it is needed to perform the process of identification of similarity to obtain the final architectural drift.

This is an important phase of our approach, because although the violations are grouped into relationship metaclass instances (`AggregatedRelationship`), the violations themselves are not following any specific grouping criteria, that is, in the way they are presented, they cannot be seen as architectural drifts but only as a set of punctual problems shown in a very fine-grained way.

To turn these fine-grained violations into architectural drifts, so that they can be easily observed in the source code, it was necessary to elaborate grouping algorithm of the correlated violations. We provided two grouping alternatives to the software architect so that he(she) can choose the one that best fits his needs. The first one is based on the Proximity Matrix and the

Table 1
Definition of terms used to calculate the evaluation metrics.

Term	Description
Ground Truth (GT)	Architectural drifts identified manually by a domain expert. Also known as *Oracle*
True Positive (TP)	Architectural drifts identified by Arch-KDM and by GT
False Negative (FN)	Architectural drifts that were not identified by Arch-KDM but were identified by GT
False Positive (FP)	Architectural drifts identified by Arch-KDM but were not identified by GT

second one based on XML hierarchy search. Fig. 5 shows these two examples. On the left side of this figure, there is a screenshot of our eclipse plugin representing the first approach, and on the right side, there is a screenshot of a UML diagram representing the second approach.

The first approach is based on a combination between the Proximity Matrix algorithm and clustering by the DBSCAN algorithm. To implement this approach, it was used the resources provided by the tool named Weka (Bouckaert et al., 2010; Garner, 1995; Borah and Bhattacharyya, 2004; Ester et al., 1996). The matrix is based on the hierarchy path of the architectural elements source and target. As can be seen on the left side of Fig. 5, each item named "Drift X" represents an architectural drift founded by the combination of these two algorithms. In this example, the algorithm generated six architectural drifts from a total of 10 violations found at the architectures comparison stage.

The second approach is named DriV-UML. It shows architectural drifts graphically using UML diagrams (classes and packages). Here, drifts are grouped by an algorithm based on the XML hierarchy of the elements in the KDM instance. It is also important to mention that in this context, the UML diagrams are used in a way they were not meant to be used. In this case, instead of showing the system as a whole, the diagrams illustrate only the architectural drifts. This strategy's main point is: each visible UML relationship represents at least one architectural drift. Also, the UML elements (classes and packages) represent the classes and architectural elements related to the drift.

4. Evaluation

In this section, we present the evaluation of our approach involving two systems of different sizes: LabSys (7.539 LoC) and FreeMind (87.000 LoC). As the goal in this paper is to present an ACC approach for KDM-represented systems, it is expected that it has a high accuracy when identifying architectural violations regardless of the fact of the system be represented as a KDM instance.

Three main factors influence the accuracy of this process: (*i*) the correct specification of the planned architecture; (*ii*) the correct mapping and extraction of the Current Architecture and (*iii*) the correct implementation of the algorithm that performs the checking. The evaluation we have conducted and showed in this section concentrates just on the checking process, focusing on showing the accuracy of the process of identifying the existing drifts. An evaluation of our DSL can be found elsewhere (Landi et al., 2017).

Our evaluation is based on the metrics precision, recall and f-measure (Makhoul et al., 1999; Landgrebe et al., 2006; Roncero, 2010; Pérez-Castillo et al., 2011). Table 1 shows four terms used to formalize these three metrics and Table 2 shows a brief description of each metric used in our evaluation.

Table 2
Metrics used in our evaluation.

Description	Metric
Precision give us the ratio between truly identified architectural drifts over the total quantity of relevant architectural drifts (a sum of ground truth and false positives)	$Precision(P) = \frac{TP}{GT+FP}$
Recall give us the ratio between truly identified architectural drifts over the total quantity of successfully architectural drifts retrieved (a sum of ground truth and false negatives)	$Recall = \frac{TP}{GT+FN}$
F-Measure is the harmonic average of the precision and recall where an F-Measure score reaches its best value at 1 (perfect precision and recall) and worst at 0	$f - measure(F) = \frac{2 \cdot P \cdot R}{P+R}$

Table 3
LabSys size metrics.

Source element count	LoC	Components	Packages
4.378	7.539	137	13

4.1. Evaluation 1: Evaluating Arch-KDM with LabSys

In this first part of the evaluation we have used a small system called LabSys (*Laboratory System*) with 7.359 LoC. It was developed in the Federal University of Tocantins (UFT), Brazil. The main purpose of this system is to manage the laboratories of the whole university.

The evaluation presented here was structured according to Wohlin guidelines (Wohlin et al., 2000). The following items summarize the evaluation:

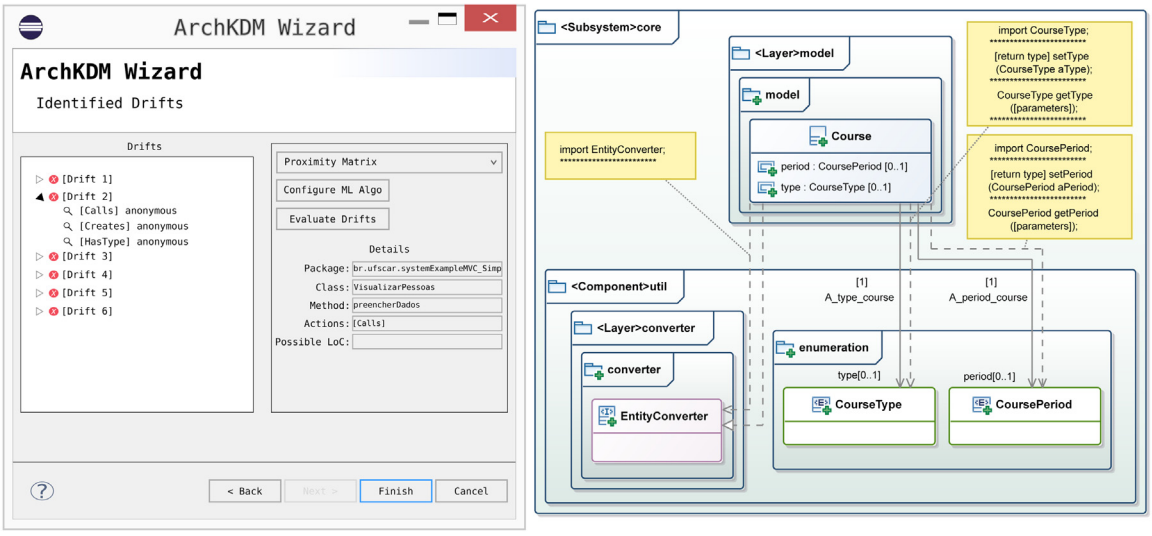
- **(I) Research Question:** Does Arch-KDM reach good levels of precision, recall and f-measure when identifying architectural drifts?
- **(II) object of study:** The Algorithms of Arch-KDM that perform the checking process;
- **(III) goal/purpose:** Checking the precision, recall and f-measure of Arch-KDM in the checking process, i.e., its performance for identifying drifts;
- **(IV) perspective:** software architects trying to find out architectural violations in systems;
- **(V) quality focus:** The accuracy of the Arch-KDM approach;
- **(VI) context:** academic context.

4.1.1. Methodology of LabSys evaluation

The methodology of this evaluation involves three steps. The first one was the manual identification of the architectural drifts of the system under analysis for creating an oracle. The second one was to apply Arch-KDM on the system to get the architectural drifts and the third one was to analyze the results.

LabSys was developed considering different architecture styles but without an explicit documented architecture. There were just a few pieces of documentation. Therefore, with the aid of the developers and analyzing the source code, it was possible to make its Planned Architecture explicit. Table 3 presents some *LabSys* metrics about its complexity. The first column shows the total number of statements added to the total number of programming elements defined in the system. The second column shows the lines of code, which indicates that the system has a medium size. The third column shows the number of classes in the system, and the last column shows the number of packages.

Step 1: Creating the oracle for the architectural drifts of LabSys



(a) Drifts based on proximity matrix

(b) Drifts based on XML hierarchy

Fig. 5. Example of our two approaches to visualize the architectural drifts.

The main goal of this step was to manually analyze the entire source code of the system in order to find architectural drifts. This analysis was executed by following three activities and checking twice each class of the system. The activities are: (*i*) Analyze each line of a class; (*ii*) if the line references to an object or item that not belongs to the own class, then we verify the called element and store the caller and called elements; (*iii*) check if the stored pair is permitted in the planned architecture. If not, we annotate it as an architectural drift. In order to perform the first activity, we created a planned architecture for *LabSys*, as we show in Fig. 6. The planned architecture is conformed by the Subsystem *core* which in turn has three Layers (*view*, *controller* and *model*) and a Component (*util*).

Layer *model* is composed by the Component *repository* and Component *util* is composed by three Layers (*validator*, *converter* and *generic*). The arrows represent the allowed access of an architectural element to one or more architectural elements. The annotation (*hierarchical access*) states that an architectural element belongs to an specific level of abstraction.

Table 4 presents the matrix with the number of architectural drifts that were identified manually (the oracle) from architectural elements of planned architecture. The values obtained to elaborate the table were removed from the manual identification of architectural drifts, as we explained before.

Notice that only were found drifts from *model* layer to *util* component and *converter* layer, from *validator* layer to *util* component and from *generic* layer to *util* component. In Table 5 we depict three lines of the oracle. The complete oracle can be access in the following link "https://goo.gl/75mssX".

Summing up, our oracle found 119 architectural drifts. The greatest amount occurs from the *validator* layer to *util* component with 52 drifts while the least amount occurs from *generic* layer to *util* component.

Step 2 - Running the Architectural Checking Process on LabSys

In this step, we applied Arch-KDM to *LabSys* in order to get the architectural drifts identified by our approach. After that, we compared the results of this step with the previous one to recognize the lines of source code defined as architectural drifts. The result of this step was the association of each architectural drift to one term of Table 1.

As our approach is for KDM-represented systems, we needed to generate a KDM instance that represents LabSys. To do that, we had to create the specification of the PA for LabSys using our DCL-KDM. Listing 2 shows the PA of our experiment and it reflects the diagram of Fig. 6. Line 2 specifies the *core* subsystem. Lines 4 – 6 specify three layers (*view*, *controller* and *model*) in hierarchical access inside of *core* subSystem. Line 7 specifies *repository* component inside of *model* layer. Line 9 specifies *util* component inside of *core* subSystem. Lines 10 – 12 specify three layers (*validator*, *converter* and *generic*) in the same hierarchical level inside of *util* component. From line 16 to 34 the PA specifies the accesses.

```
1  architecturalElements{
2      subSystem core;
3
4      layer view, level 3, inSubSystem: core;
5      layer controller, level 2, inSubSystem: core;
6      layer model, level 1, inSubSystem: core;
7      component repository, inLayer: model;
8
9      component util, inSubSystem: core;
10     layer validator, level 1, inComponent: util;
11     layer converter, level 1, inComponent: util;
12     layer generic, level 1, inComponent: util;
13
14     }restrictions{
15
16     controller can-depend repository;
17     controller can-depend model;
18
19     only controller can-depend util;
20     only controller can-depend validator;
21     only controller can-depend converter;
22     only controller can-depend generic;
23
24     only util can-depend model;
25     only util can-depend repository;
26
27     only validator can-depend model;
28     only validator can-depend repository;
29
30     only converter can-depend model;
31     only converter can-depend repository;
32
33     only generic can-depend model;
34     only generic can-depend repository;
35 }
```

Listing 2: Specification of the Planned Architecture of LabSys

Step 3: Validation of results

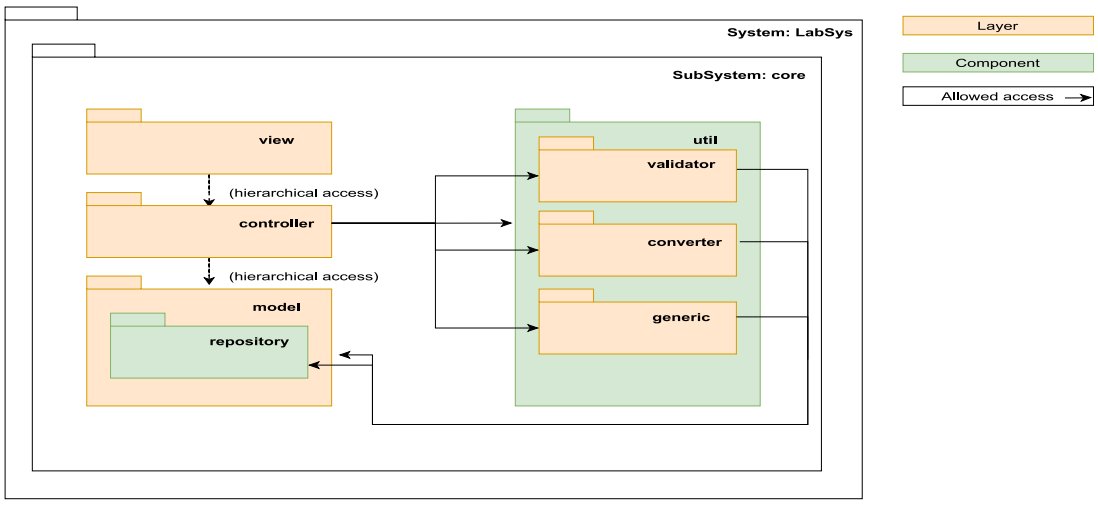


Fig. 6. Planned Architecture of LabSys.

Table 4
Results of the manual identification of architectural drifts from *LabSys*.

			View			Controller			Repository			Model			Util			Validator			Converter			Generic			
			0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	**1**	0	0	0	0	0	0	0	0	0	**Calls**
			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Uses type**
			0	0	0	0	0	0	0	0	0	0	0	0	0	**16**	0	0	0	0	0	0	0	0	0	0	**Creates**
			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Extends**
Model	Validator	Generic	0	0	0	0	0	0	0	0	0	0	0	0	0	**8**	0	0	0	0	**12**	0	0	0	0	0	**Implements**
			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Has value**
			0	0	0	0	0	0	0	0	0	0	0	0	**10**	**19**	**1**	0	0	0	**12**	0	0	0	0	0	**Imports**
			0	0	0	0	0	0	0	0	0	0	0	0	**30**	**9**	0	0	0	0	0	0	0	0	0	0	**Has type**
Total			0	0	0	0	0	0	0	0	0	0	0	0	**41**	**52**	**2**	0	0	0	**24**	0	0	0	0	0	**119**

Table 5
Oracle example of *LabSys*.

Source element	Line	Target element	Line	Type
`Block.java`	3	`EntityConverter.java`	N/A	Import a class
`Campus.java`	20	`EntityConverter.java`	N/A	Implement a class
`Course.java`	98	`CoursePeriod.java`	N/A	Method return

In this step, we applied the Jury Method (da Fonseca et al., 2007; Matos, 2014) to evaluate the interpretation of the domain expert when performed the oracle and the comparisons of architectural drifts that were manually generated in the previous step. This method consists of the use of a jury composed of judges that individually analyze the results proposed by the domain expert. After the analysis of the judges, it was verified if they agreed. It was done by using the Percentage of Absolute Agreement (PAA) (Matos, 2014). This calculation is performed by dividing the number of times that judges agree over the total quantity of evaluated items by them. For some authors, the minimum acceptable value is 75%, and a value of 90% is considered high. In order to perform the Jury Method, we follow 4 activities.

In the first activity, we chose 4 volunteers to compose the jury table. Three volunteers performed evaluations and one volunteer acted as solve problem man in case of disagree among the other judges. All four volunteers were domain experts in computing and software development. Their names are not disclosed by privacy issues but their profiles comprises: (*i*) a software developer with an MSc in software engineering with experience in databases and working in a worldwide aviation enterprise; (*ii*) a software developer with an MSc in computer science with experience in software modernization; (*iii*) a privacy expert and professor from Pontifical Catholic University of Minas Gerais, Brazil; (*iv*) a software developer with a Ph.D. in computer science with experience in software modernization.

In the second activity, we delivered the Consent Form Free and Informed to the judges for acceptance purposes and to declare the role of each participant. In the third activity, we delivered the necessary documentation to the judges for their participation, such as the analyzed source code, the oracle developed by the domain expert and the results obtained by Arch-KDM. In the fourth activity, we requested the jury to do an analysis of the source code of the system and to compare it with the planned architecture. Thus, they were able to classify the architectural drifts found by the domain expert and Arch-KDM. After their evaluation, the domain expert had the task of evaluation of three items in two categories, "Agree" if he approves the evaluation performed by the jury and "Not agree" if he does not approve the evaluation performed by the jury. The defined items are: (*i*) architectural drifts manually found; (*ii*) identification of which architectural drift that was manually found represents the architectural drift automatically found and; the evaluation as a whole.

Step 4: Results of ACC using LabSys

Table 6 represents a matrix contains the result of the third step of the LabSys evaluation. Each cell shows the amount of deviation – automatically detected – between an architectural element and a specific type of drift. ArchKdM was able to detect 100 architectural drifts, but after analyzing the data, the real value was 94 drifts. This happened because our clustering algorithm clustered six drifts that were already identified in two groups due to the proximity coefficient value set up. Nevertheless, as it was the only divergence, we opted to maintain the value.

In Table 7, the result of the fourth step of the evaluation can be observed. In Table 7, it is possible to observe that the jurors numbered from 1 to 3 agreed and validated the execution of the

Table 6
Automatic identification of the results of architectural deviations from *LabSys*.

	View			Controller			Repository			Model			Util			Validator			Converter			Generic			
Model Validator Generic	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	**1**	0	0	0	0	0	0	0	0	0	**Calls**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Uses type**
	0	0	0	0	0	0	0	0	0	0	0	0	0	**16**	0	0	0	0	0	0	0	0	0	0	**Creates**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Extends**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**12**	0	0	0	0	0	**Implements**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**Has value**
	0	0	0	0	0	0	0	0	0	0	0	0	**10**	**11**	**1**	0	0	0	**12**	0	0	0	0	0	**Imports**
	0	0	0	0	0	0	0	0	0	0	0	0	**30**	**6**	0	0	0	0	0	0	0	0	0	0	**Has type**
Total	0	0	0	0	0	0	0	0	0	0	0	0	**41**	**33**	**2**	0	0	0	**24**	0	0	0	0	0	**100 (94)**

Table 7
Judges score by evaluation question.

Rated item	Judge 1	Judge 2	Judge 3
Manual identification	Agree	Agree	Agree
Architectural drifts: Manual × Auto.	Agree	Agree	Agree
Evaluation	Agree	Agree	Agree

Table 8
Values of the evaluation metrics.

Term	Value
Ground Truth (GT)	119
True Positive (TP)	94
False Negative (FN)	25
False Positive (FP)	0

Table 9
Values of the metrics.

Values
Precision (P) = 0,7899
Recall (R - *Recall*) = 0,6527
f-measure (F) = 0,7147

Table 10
Precision level scale for information retrieval.

Precision	Level
Precision < 0.47	Very low
0.47 < Precision < 0.56	low
0.56 < Precision < 0.63	Average
0.63 < Precision < 0.72	High
0.72 < Precision	Very High

evaluation. In this way, it was possible to carry out the last step of the evaluation.

As mentioned before, we used the metrics precision, recall and F-measure. In Table 8 can be observed the values for each of the four terms necessary for the calculation of the metrics. Therefore, substituting the values of Table 8 in the metrics of Table 2 we obtain our final results. Considering the architectural deviations recovered (TP), the value of the precision is 0,7899 representing a precision of 78,99%. Using the values stipulated and proposed by Perez-Castillo et al. (2011) (Table 10), it can be affirmed that the technique and the computational support developed can be used with a certain degree of confidence since it reached the very high precision level according to the authors (see Table 9).

The recall calculation is the ratio of deviations found on all deviations and false negatives. As noted, the recall value is 0.6527 representing 65.27%. This means that of all architectural deviations, 65.27% were recovered. Finally, we have the metric f-measure used to evaluate the accuracy by performing a weighting between the values of precision and recall. The result value is 0.7147 representing 71.47%. This metric is an indication of performance so that the closer to the total is the result, that is, 100%, the better the performance of the object under analysis (Roncero, 2010).

Analyzing the obtained metrics were possible to affirm that the approach and the computational support developed have positive and very promising results. Analyzing in-depth the architectural deviations not found by the computational support, it was possible to observe that the algorithm contains some faults regarding code elements represented in the KDM Code package. In the specific case of this evaluation, it was observed that the deviations not found automatically were because they are elements of the composition of the TemplateUnit metaclass. It is believed that by evolving the algorithm for the other unrecognized elements, it is possible to reach an accuracy of 100%.

Another interesting point to be mentioned in that we made contact with the original developers of LabSys showing the results we have reached. They considered very relevant to know about the existence of those drifts, but they cannot guarantee that the maintenance problems they are suffering is consequence of these drifts. Hence, they recognize the drifts found are clear problems that must be solved, even though most of them being related to the util package.

Table 11
Freemind size metrics.

Source element count	LoC	Components	Packages
37.356	84.357	438	45

4.2. Evaluation 2: Evaluating ArchKDM with FreeMind

This evaluation was conducted with a medium-size system called FreeMind (84 KLoc). It is a system written in Java used to create mind-mapping and it was used by Pruijt et al. (2017) for comparing ten (10) different ACC approaches. Just like (Pruijt et al., 2017), we used the version 0.9.0 of FreeMind for our evaluation, that is available here.[3] Table 11 shows some metrics of FreeMind.

As mentioned, Pruijt et al. (2017), performed a comparison among ten (10) different ACC approaches using the FreeMind system. However, unlikely the LabSys evaluation, in this case both the Planned Architecture and the Oracle (containing the existing drifts that violate the rules prescribed in the PA) were available. Therefore, we decided to apply Arch-KDM in this system and compare the results with the oracle presented by the authors. Obviously, although the PA was available in the paper, it was not specified using our DSL. Thus we have to build the specification for enabling to use ArchKDM.

The evaluation presented here was also structured according to Wohlin guidelines (Wohlin et al., 2000). The following items summarize the evaluation:

[3] http://freemind.sourceforge.net/wiki/index.php/Download.

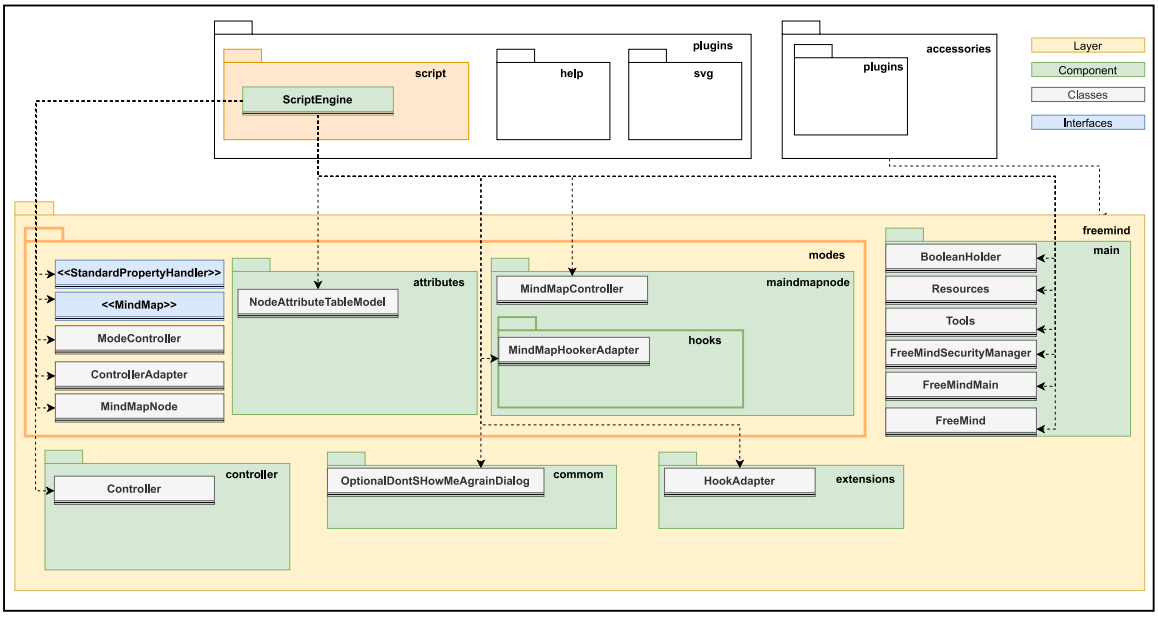


Fig. 7. Structure of Freemind with dependency relations.

- **(I) object of study:** The Algorithms of Arch-KDM that perform the checking process;
- **(II) goal/purpose:** To assess the performance of the Arch-KDM to identify other types of architectural drifts that were not found on the LabSys;
- **(III) perspective:** Of academics and practitioners that uses the KDM in the **context** of modernization systems;
- **(IV) quality focus:** The **focus** is on precision, recall, and f-measure;
- **(V) context:** Academic context.

4.2.1. Methodology of FreeMind evaluation

The methodology used in this evaluation had three steps. The first one was specifying the Planned Architecture presented by Pruijt et al. (2017) using our DSL. The second was to apply the Arch-KDM on the FreeMind system to find the architectural drifts. The third one was to analyze the results. The next subsections detail the methodology steps.

Step 1 - Specifying the planned architecture. Pruijt et al. (2017) present the following rule as the Planned Architecture of FreeMind: **the class *plugins.script.ScriptingEngine* is not allowed to use any from package *freemind***. In other words, any dependence from class *ScriptingEngine* to any element within package *FreeMind* is considered a drift.

Fig. 7 shows the current architecture of FreeMind with some dependencies — not all are shown for not polluting the diagram. Each dependency line represents one or more concrete dependencies. For example, although there is just one dependency line between the classes *ScriptingEngine* and *MindMapController*, there may be two or more method calls between these classes. Some classes and packages was also omitted to better visualization. Therefore, we concentrate on showing only the dependencies between the class *ScriptingEngine* and package *FreeMind* as these are the most important from the point of view of the PA .

There are two higher level layers in the architecture, represented by the packages script and FreeMind. Inside the script layer it is shown just the class *ScriptingEngine*, as it belongs to the rule prescribed by the PA. As can be seen, there are 17 (seventeen) dependencies from the *ScriptingEngine* class to others of the package *FreeMind*: *Controller, ControllerAdapter, FreeMind, FreeMindMain, FreeMindSecurityManager, HookAdapter, MindMap, MindMapController, MindMapHookAdapter, MindMapNode, ModeController, NodeAttributeTableModel, OptionalDontShowMeAgainDialog, StandartPropertyHandler, Resources, Tools and BooleanHolder*. Therefore, Pruijt et al. (2017) state that there are 17 architectural violations/drifts in this system that should be find out by an ACC approach.

As we have done with LabSys, we also had to create a specification of the PA of FreeMind for generating a KDM instance representing this system. Listing 3 shows the PA we have specified for FreeMind.

```
1  architecturalElements{
2      subSystem freemindSystem;
3
4      layer freemind, level 1, inSubSystem: freemindSystem;
5      layer plugins, level 1, inSubSystem: freemindSystem;
6
7      component script, inLayer: plugins;
8
9      component main, inSubSystem: freemind;
10     component controller, inLayer: freemind;
11     layer modes, level 1, inLayer: freemind;
12     component extensions, inLayer: freemind;
13     component common, inLayer: freemind;
14     component mindmapnode, inLayer: modes;
15     component attributes, inLayer: modes;
16     component hooks, inLayer: modes;
17 }restrictions{
18     script cannot-depend freemind;
19 }
```

Listing 3: Specification of the Planned Architecture of FreeMind

Line 2 specifies the *freemindSystem* sub-system. Lines 4 — 5 specify two layers (FreeMind and plugins) inside *freemindSystem*. Line 7 specifies a component (script) inside layer plugins. Lines 9 — 10 and 12 — 13 specify 4 components inside *freemindSystem*. Line 11 specifies e layer inside *freemindSys*. Lines 14 — 16 specifies 3 components inside *modes* layer. Line 18 specifies the unique restriction rule.

Notice that the granularity level of architectural violations considered by Pruijt is different from ours. They work in a higher abstraction level, considering as violations/drifts accesses in classes and interfaces. For example, it does not matter if the

Table 12
Tools assessed by Pruijt et al. (2017).

Techniques	ConQAT	Dependometer	dTrangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	structure 101
Text-based		×				×		×		
Dependency Structure Matrix (DSM)			×		×					
Reflexion Model	×			×			×			
Diagram-based									×	×

Table 13
Results of the architectural violations identification for FreeMind: 1 = detected and 0 = not detected.
Source: Adapted from Pruijt et al. (2017).

Reported classes	Nr of Dep.	Arch-KDM	ConQAT	Dependometer	dTrangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	structure 101
Controller	1	1 (1)	1	1	1	1	1	1	1	1	1	1
ControllerAdapter	5	1 (1)	0	0	0	1	0	0	1	0	0	0
FreeMind	12	1 (1)	0	1	0	1	0	0	1	0	1	0
FreeMindMain	16	1 (10)	1	1	1	1	1	1	1	1	1	1
FreeMindSecurityManager	5	1 (2)	1	1	1	1	1	1	1	1	1	1
HookAdapter	6	1 (2)	0	0	0	1	0	0	1	0	0	0
MindMap	1	0	1	1	1	1	1	1	1	1	1	1
MindMapController	6	1 (4)	1	1	1	1	1	1	1	1	1	1
MindMapHookAdapter	5	1 (3)	1	1	1	1	1	1	1	1	1	1
MindMapNode	17	1 (6)	1	1	1	1	1	1	1	1	1	1
ModeController	2	1 (2)	1	1	1	1	1	1	1	1	1	1
NodeAttributeTableModel	6	1 (3)	1	1	1	1	1	1	1	1	1	1
OptionalDontShowMeAgain-Dialog	5	1 (1)	1	1	1	1	1	1	1	1	1	1
StandardPropertyHandler (inner)	1	0	0	1	0	1	0	1	0	1	1	1
Resources	2	1 (2)	1	1	1	1	1	1	1	1	1	1
Tools	6	1 (2)	1	1	1	1	1	1	1	1	1	1
BooleanHolder (inner)	13	1 (9)	0	1	0	1	0	1	0	1	1	1
Number of classes or interfaces identified		15	12	15	12	17	12	14	15	14	15	14
Recall (in%)		88	71	88	71	100	71	82	88	82	88	82

class ScriptingEngine had one or many dependencies to another class X, they will count as only 1 violation, i.e., the cardinality of the dependency is not important. In our case, we would count every dependency as a violations. Therefore, in order to allow a fair comparison, in this part of evaluation we adopt the same counting strategy as Pruijt.

Step 2 - Running the Architectural Checking Process on FreeMind. Likewise we have done with LabSys, we have also applied Arch-KDM for identifying the drifts of FreeMind. The steps are not detailed here but follows the same already presented earlier. Initially we have used our Wizard for mapping the abstractions declared in the PA to the source-code of the system. After that, we could trigger the checking process and get the results.

Step 3 - Results of ACC using Freemind System

As previously said, this second part of evaluation was conducted with the support of the existing work of Pruijt et al. (2017). In that paper the authors compare 10 (ten) ACC Approaches using the FreeMind System and present the recall of each ACC approach. Therefore, it was a great opportunity of comparing several approaches with Arch-KDM.

Table 12 shows all tools evaluated by Pruijt et al. and results of each. Pruijt et al. (2017). As can be noted, the evaluated tools support different techniques (first column) to identify violations. For instance, Dependometer, Macker, and Sonar Architecture Rule Engine (Sonar ARE) are text-based tools. On the another hand, dTangler and Lattix rely on Dependency Structure Matrix (DSM). Also, ConQAT Architecture Analysis, JITTAC, and SAVE are based on the Reflexion Model technique. Lastly, Sonargraph and Structure101 are diagram-based.

Table 13 was adapted from Pruijt et al. (2017) and shows the results for each of the 10 tools tested by them plus the results we have obtained for Arch-KDM. We have opted for leaving all the tools tested by them so that we can compare the results of Arch-KDM with the others. The first column shows the name of 17 (seventeen) classes/interfaces from which the class *ScriptingEngine* depends on. All of these classes are within the freemind package, so according to the rule of the PA presented by the authors, these 17 elements should be identified as violations/-drifts. The column Nr. of Dep. indicates the number of existing dependencies from *ScriptingEngine* to the respective class. For instance, ConQAT has identified at least 1 dependency between *ScriptingEngine* and *FreeMindMain*. Conversely, ConQAT did not identify any dependency between these two classes.

In Table 13, for each ACC Approach, the number 1 indicates it was able to identify that violation and 0 indicates the opposite. The last line of the table is the recall value for each of the approaches. Arch-KDM obtained better results than 6 (six) approaches (ConAQT, dTrangler, Lattix, Macker, Sonar ARE and structure 101); equal results for three approaches (Dependometer, SAVE and Sonoargraph) and worse results than JITAC. That is, ArchKDM was not able to identify two dependencies:

(a) *ScriptingEngine* $\longrightarrow$ *MindMap* and;
(b) *ScriptingEngine* $\longrightarrow$ *StandardPropertyHandler*

This was expected because at this moment Arch-KDM does not take into account indirect/transitive dependencies. An indirect dependency is a dependency in the from-class of which the to-class cannot be determined without the analysis of the code of another class (Pruijt et al., 2017). In general, a dependency relation is indirect, when the dependency exists transitively through an intermediate module. For example, in dependency (a), *MindMap* is a java interface and a way to determine if there is an indirect dependency between *ScriptingEngine* and *MindMap* is by analyzing the direct dependency between *ScriptingEngine* and *MindMapNode*. In turn, *MindMapNode* has a direct dependency with *MindMap* through *MindMapgetMap*() method declaration. Therefore, there is an indirect dependency between *ScriptingEngine* and *MindMap*.

Table 14
Accuracy of Arch-KDM in freemind evaluation.

Precision	Recall	F-measure
100%	88.2%	93.7%

In dependency (b), *StandardPropertyHandler* is an inner class that belongs to the *OptionalDontShowMeAgainDialog* class. In this case *ScriptingEngine* call *StandardPropertyHandler* by instantiating it by using the operator *new* in the following manner: *new OptionalDontShowMeAgainDialog.StandardPropertyHandler(..)*. Therefore, Arch-KDM only identifies the *OptionalDontShowMeAgainDialog* class as a dependency.

Regarding the precision, recall, and f-measure, we calculated these metrics taking into accounts the dependencies between *ScriptingEngine* and the 17 reported classes (Pruijt et al., 2017) of package *freemind*. In this case, the Arch-KDM should have reported only the dependencies (drifts) with these 17 classes (see in Table 13). If the Arch-KDM reported any dependency with another class of package *freemind*, this was to consider a false positive (FP). Likewise, if the Arch-KDM did not report an expected dependency, this was considered a false negative (FN). A true positive (TP) occurs when the Arch-KDM reported an expected dependency and, a false positive (FP), in this evaluation, is always 0 because according to Pruijt et al. (2017), the *ScriptingEngine* depends on only these 17 reported classes.

Table 14 shows the values(in %) of precision, recall and f-measure. As noted, the recall value is 88.2%. This indicating that of all dependencies between *ScriptingEngine* and package *freemind* our approach was able to identity 88.2% of them. This is a high value, indicating that our approach performs well to identify the positive instances of architecture drifts. Regarding precision, we achieved 100%. However, we believe that this occurs because the oracle does not contain false positives . Lastly, the f-measure value is 93.7%, indicating that the Arch-KDM performs so that the closer to the total is the result.

Table 15 shows the type of drift found by Arch-KDM. The first column shows the reported classes, the last column represents the total of dependency and, the rest columns show the type of drift. For instance, *ScriptingEngine* depends on *MindMapController* and makes 2 Calls, 1 HasType and 1 Import - 4 in total (last column).

As can be noted in Table 15, Arch-KDM was able to identify drift of type Extends, that was not identified in LabSys evaluation. The main goal here is to show that our approach was able to identify this type of architecture drifts. That is, if Arch-KDM can identify at least one instance of a drift, it can be used to detect other instances in the system.

4.3. Fine-grained analysis of drifts in freemind

In this subsection, we perform a fine-grained analysis of the drifts identified by Arch-KDM in Freemind. As there are several cases to be analyzed we will take a specific one as an example; *ScriptingEngine* $\longrightarrow$ *FreeMindMain*.

Table 13 shows the number of dependencies between *ScriptingEngine* and several classes/interfaces. Particularly, according to Pruijt et al. (2017) the relationship *ScriptingEngine* $\longrightarrow$ *FreeMindMain* has 16 dependencies. We analyzed the *ScriptingEngine* class to corroborate this number, which is correct and we also verified why Arch-KDM could not identify these dependencies. Table 16 shows the fine-granular dependencies between *ScriptingEngine* and *FreeMindMain* classes.

The first column depicted with a "X" represent all dependencies identified by Arch-KDM. The second column shows the code element of *ScriptingEngine* class that is responsible for the existence of the relationship. The third column shows the code element of *FreeMindMain* class that receives the relationship. The fourth column presents the type of the relationship and the fifth column depicts the line number where occurs the relationship in the source-code.

The result of our analysis indicates that Arch-KDM did not identify the six drifts due to a lack of representation of the KDM instance. For example, Listing 4 shows the code that corresponds to the line 156 of *ScriptingEngine*. It is an assignment statement that involves a "new" operator for instantiating the *OptionalDontShowMeAgainDialog* class. The constructor of this class receive several parameter where one of them is a method call *frame.getJFrame()* (lines 2 – 3).

Listing 5 shows the corresponding code of Listing 4 in the KDM instance that was generated by MoDisco. The code relation of line 6 of type HasValue indicates that there is a relationship to an "Element method invocation" but the KDM instance does not implement the assignment.

Finally, the recall in percentage of Arch-KDM by taking into account all fine-grained dependencies is $\frac{50}{109} \approx 46\%$. Although this number is not quite so good, it is heavily influenced by the completeness of the KDM instance. Therefore, if the KDM instance truly can represent the whole source-code the accuracy of Arch-KDM will be incremented notoriously. However, another possibility is to change KDM in order to represent this part.

4.4. Threats to validity

We can list the following threats:

- Regarding the PA created for Freemind evaluation. Someone could argue the package *freemind* should have mapped and the unique rule should be: *script cannot-depend freemind*. We claim that this does not affect the evaluation because all the packages that *ScriptingEngine* depends on were defined in the PA. So, the Arch-KDM should report the same violations.
- The granularity of Freemind evaluation. Someone could argue the results of Arch-KDM cannot be compared with Pruijt et al. (2017) because is not in the same level of granularity, i.e., those authors group all the fine grained dependencies in just one, represented by a dependency between classes. However, the comparison is indeed fair in our point of view. All the others approaches presented by those authors in Table 13 were also analyzed by grouping fine-grained dependencies. When the authors present that some approach has identified the drift, it is not presented if the approach was able to find out all the existing dependencies or just one of them. Therefore, we did the same.
- The use of just two systems and the complexity of them. Someone could argue that the ideal would be conducting the evaluation with more than two systems. However, we claim that this does not affect the evaluation so much. This happens because the systems to have the conformance checked are represented as KDM instances — there is a standard representation way. If Arch-KDM has been able to identify a specific **type** of drift (for example, method calls), it will be able to detect this same type of drift in any KDM-represented system. Therefore, we could concentrate in systems having other types of drifts rather than the ones presented in LabSys and Freemind;
- Regarding to the process of building the oracle of LabSys evaluation. Someone could also argue the way the oracle was built benefited the identification process. To minimize this threat we have followed a three-step process as described in step 1 of Section 4.2.1;

Table 15
Number of drifts identified by Arch-KDM per type.

Classes which ScriptingEngine depends on	Calls	Creates	Extends	HasType	Implements	Imports	UsesType	Total
Controller	1	0	0	0	0	0	0	1
ControllerAdapter	1	0	0	0	0	0	0	1
FreeMind	0	0	0	0	0	1	0	1
FreeMindMain	8	0	0	1	0	1	0	10
FreeMindSecurityManager	0	0	0	1	0	1	0	2
HookAdapter	3	0	0	0	0	0	0	3
MindMap	0	0	0	0	0	0	0	0
MindMapController	2	0	0	1	0	1	0	4
MindMapHookAdapter	1	0	1	0	0	1	0	3
MindMapNode	0	0	0	5	0	1	0	6
ModeController	2	0	0	0	0	0	0	2
NodeAttributeTableModel	1	0	0	1	0	1	0	3
OptionalDontShowMeAgainDialog	0	0	0	0	0	1	0	1
StandardPropertyHandler (inner)	0	0	0	0	0	0	0	0
Resources	2	0	0	0	0	0	0	2
Tools	1	0	0	0	0	1	0	2
BooleanHolder (inner)	3	1	0	4	0	1	0	9
Total	25	1	1	13	0	10	0	50

Table 16
Dependencies between *ScriptingEngine* and *FreeMindMain*.

	From	To	Type	Line
X	plugins.script.ScriptingEngine	freemind.main.FreeMindMain	Imports	41
X	plugins.script.ScriptingEngine.performScriptOperation	freemind.main.FreeMindMain.setWaitingCursor	Calls	98
X	plugins.script.ScriptingEngine.performScriptOperation	freemind.main.FreeMindMain.setWaitingCursor	Calls	125
X	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain	HashType	153
	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.getFrame	Calls	156
	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.getProperty	Calls	195
	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.getProperty	Calls	197
	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.getProperty	Calls	199
	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.getProperty	Calls	201
	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.getProperty	Calls	203
X	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.setProperty	Calls	267
X	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.setProperty	Calls	271
X	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.setProperty	Calls	275
X	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.setProperty	Calls	279
X	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.setProperty	Calls	282
X	plugins.script.ScriptingEngine.executeScript	freemind.main.FreeMindMain.getResourceString	Calls	324

```
1 int showResult = new OptionalDontShowMeAgainDialog(frame
2                     .getJFrame(), pMindMapController.getSelectedView(),
3                     "really_execute_script", "confirmation",
4                     pMindMapController,
5                     new OptionalDontShowMeAgainDialog.StandardPropertyHandler(
6                             pMindMapController.getController(),
7                             FreeMind.RESOURCES_EXECUTE_SCRIPTS_WITHOUT_ASKING),
8                     OptionalDontShowMeAgainDialog.ONLY_OK_SELECTION_IS_STORED)
9                     .show().getResult();
```

Listing 4: Code Snippet of the Drift of Line 156

```
1 <codeElement xsi:type="code:StorableUnit" name="showResult" type="//@model.0/@codeElement.5/@codeElement.0" kind="local">
2             <attribute tag="export" value="none"/>
3             <source language="java">
4               <region file="//@model.2/@inventoryElement.415" language="java"/>
5             </source>
6             <codeRelation xsi:type="code:HasValue" to="//@model.1/@codeElement.4859" from="//@model.0/@codeElement.3/
                  @codeElement.3/@codeElement.1/@codeElement.8/@codeElement.1/@codeElement.1/@codeElement.1/@codeElement
                  .0/@codeElement.0"/>
7             <codeRelation xsi:type="code:HasType" to="//@model.0/@codeElement.5/@codeElement.0" from="//@model.0/
                  @codeElement.3/@codeElement.3/@codeElement.1/@codeElement.8/@codeElement.1/@codeElement.1/@codeElement
                  .1/@codeElement.0/@codeElement.0"/>
8 </codeElement>
```

Listing 5: Code Snippet of Drift of Line 156 in KDM

- The reliance on Software Engineers to evaluate the results.

  Although the results of LabSys were evaluated through the application of the judging method, as is possible in evaluations with humans, the results may have been affected by some degree of subjectivity (construction validity).

5. Related works

5.1. Works related to DCL-KDM

Currently, there are several strategies to specify and serialize a Planned Architecture in ACC approaches. Nevertheless, the majority employs Architectural-Description Languages (ADL) for its specification and proprietary metamodels for its serialization (Stafford, 2001; Aldrich et al., 2002; Sneed, 2005; Sangal et al., 2005; Abi-Antoun and Jonathan, 2008; Duszynski et al., 2009; Deissenboeck et al., 2010; Adersberger and Philippsen, 2011; Herold and Rausch, 2013; Maffort et al., 2016). Unfortunately, the use of proprietary metamodels hinders interoperability between tools. Besides, most of the existing ADLs provide just modules, ports and connectors (Hussain, 2013), and none of them focus on automatic generation of rules for strict layering and composition. So they would only use a small part of the KDM. To the best of our knowledge, only our research group in this paper and also in another project (San Martín and Camargo, 2021), have dedicated efforts for specifying planned architectures in KDM and also considering pre-defined architectural rules. In this way, the related works present some support for specifying PAs; graphical or textual.

SAVE (Duszynski et al., 2009) 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 (Sangal et al., 2005) 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 (Stafford, 2001) 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 (Adersberger and Philippsen, 2011) defines the traceability of UML component models to code using AOP type pattern expressions. Herold and Rausch (2013) 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.

ArchJava and ArchLint rely on AST as the underlying model for performing ACC. ArchJava (Aldrich et al., 2002) extends Java with architectural modeling constructs that seamlessly unify software architecture with implementation, ensuring that the implementation is according to the architectural constraints. ArchLint (Maffort et al., 2016) is a data mining approach for ACC that identifies architectural violations based on a combination of static and historical source code analysis.

In San Martín and Camargo (2021) we present a DSL for specifying PAs in the context of Adaptive Systems. Our language provides adaptive-systems specific abstractions so that engineers can be more precise when describing the planned architecture.

5.2. Works related to ACC

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 (Terra and Valente, 2009), ArchJava, and ArchLint rely on AST (Abstract Syntax Tree) as the underlying model for performing ACC. DCL employs static analysis for identifying the structural dependencies that do not respect the rules specified in the PA. ArchJava extends Java with architectural modeling constructs that seamlessly unify software architecture with implementation, ensuring that the implementation is according to the architectural constraints. ArchLint 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 proper 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 (Deissenboeck et al., 2010), SAVE (Knodel and Popescu, 2007; Duszynski et al., 2009), and SotoArch/Sotograph (Bischofberger et al., 2004) rely on graphs as the underlying model to perform ACC. ConQAT 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 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/Sotograph 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 metamodel (KDM) to represent the PA and CA. It means that researchers who are familiar with 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 (Abi-Antoun and Jonathan, 2008), ReflexML (Adersberger and Philippsen, 2011), and Herold and Rauschs approach (Herold and Rausch, 2013) rely on MDE models to perform ACC. ArchConf generates a conformance view and computes metrics between two C&C (component and connector) views. They represent various languages of the system in metamodel form of the relevant source artifacts at the desired level of detail. ReflexML defines the traceability of UML component models to code using AOP type pattern expressions. Herold and Rausch express architectural rules as formulas on a common ontology where 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, relies on KDM, which provides metaclasses to represent architectural elements and allows source code elements to be represented with one-to-one precision.

Other ACC approaches: LDM (Sangal et al., 2005) relies on Dependency Structure Matrices (DSM) 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, it is not an architecture specification independent of the system's implementation. In the dynamic analysis research line, DiscoTect (Yan et al., 2004) 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 (Çiraci et al., 2012) is a run-time verification approach for detecting inconsistencies between the dynamic behavior of

the documented architecture and the actual run-time behavior. 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

This paper presented the Arch-KDM approach, an Architectural-Conformance Checking approach to be used in the context of Architecture-Driven Modernization (ADM). All the steps of the approach rely on the KDM, which is a platform and language-independent metamodel from OMG. The tooling support we have developed is composed by:

- A Domain-Specific Language (DSL) called DCL-KDM. It allows the specification of Planned Architectures using terms like layer, component and module. This DSL also allows specifying the communication restrictions among these architectural abstractions. An important characteristic here is that the PA is serialized like a KDM instance;
- A Wizard that allows the extraction of the Current Architecture (CA) of the legacy system. To do that, the user must deliver to the Wizard a system represented in KDM. In a similar manner, the CA is also serialized as a KDM instance;
- An engine that compares both architecture representations (PA and CA) in order to detect the drifts. As the PA and CA are represented as KDM instances, all the algorithms of the engine can work over the same taxonomy, making them clearer since they work on the same metamodel.

It is important to clarify that our approach works exclusively on the identification of fine-grained drifts such as method calls, implementation, inheritance relationships, object creation, type cast or conversions, and package imports. Higher-level drifts are not detected by our approach, for instance, the absence of architectural abstractions.

Our approach checks the conformance of systems implemented in any language. This check is possible because by using our Wizard, we can recovery the CA from systems represented in KDM, so the engine is independent of specific languages and platforms. After this step, all the checking algorithms work over KDM representations. Besides, as our algorithms are based on an ISO pattern, they have a high potential for reuse. This does not happen when algorithms are developed over a proprietary or language-dependent model.

A meaningful discussion here is regarding the suitability of KDM for representing software architecture. The Code Package can represent low-level details; however, the quality of the Structure package is harder to evaluate. For example, although the Structure package has the most conventional metaclasses for representing architectural details, it lacks some other important ones, such as: Filters, Connectors, Ports, and Required/Provided Interfaces. It is worth to notice that we did not extend KDM to represent architectural details; thus, we have worked just with the existing KDM abstractions.

In the last step of our approach, both KDM representations (PA and CA) are compared, and a list of architectural violations is obtained. As both system representations are instances of the same metamodel, the algorithm (see Algorithm 2) becomes clearer, easier to understand, and, as a consequence, easier to maintain, reuse, and evolve. The usage of KDM does not impact the process quality, mainly because the Code metamodel is in an abstraction level very similar to the source code. Thus, every detail to perform an architectural checking, 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 here represent a significant portion of all architectural deviation that occurs in reality

We are aware that efficacy of the ACC proposed process depends on the completeness of the PA specification and the correct mapping with the source code. Besides, currently the process is not automatically triggered, so software architects would have also to consider including a new step in the life cycle of the system. The first point can be amended by establishing a systematic procedure for checking the specifications/mappings in order to observe whether changes/updates must be performed. Although this can bring a burden for software architects, we believe the frequency of that is not high. The second point can easily be implemented in order to make the process automatic, for example, triggering the checking process at every commit. Clearly, this alternative could make the proposal more reasonable to be adopted by the industry, where the pressure for shorter delivery periods are intense.

Besides the approach by itself, another theoretical contribution is the formalization of the terms architectural drift and architectural violation. Herein, architectural drift is what software engineers are looking for in order to fix the problem. Many times this is a line of source code. This formalization was necessary in consequence of the way KDM represents source code details. The Code Package represents source code at a low abstraction level, so a single line of source code is represented as several metaclass instances. As a result, many times, a single instance is not representative or useful from the architect's point of view. So, we decided to call "violation" each metaclass instance that can be involved in an architectural drift. Therefore, an architectural drift for us is a set of architectural violations. An essential point of this end-to-end approach is the demonstration that a complete ACC approach can be conducted using just the KDM metamodel, which differs from other approaches found in literature, as can be seen in the related works section. However, for representing Planned Architectures with KDM we have to represent the access rules between architectural elements using the KDM Aggregated Relationship, i.e., the absence of a specific type of relationship inside the aggregated means denied access. This is a particular way of using this kind of KDM relationship, but it was very useful in this context. We expect the research community publishes more evidences of KDM, so that ADM can genuinely become the de-facto standard way of conducting software systems modernization.

Regarding the scalability of our approach, it depends the most on the capacity of the tool used for generating the KDM for big systems. Currently, Modisco takes some time for representing big systems as KDM instances. However, once we have this instance at hands, the process of checking the conformance is not time-consuming. Another related point is regarding the complexity of the PA for big systems, but again, this depends on the level of detail that will be employed.

Although this work present Arch-KDM as an Architecture-Conformance Checking solution to be used in ADM (Architecture-Driven Modernization) context, it is not restricted to it. That is, Arch-KDM can be used in any situation in which engineers need to identify the drifts of the system. However, as KDM is introduced by OMG as a standard way for representing systems in ADM context, it is in this context that Arch-KDM makes more sense. This happens because the architecture-conformance checking process is just one of the activities of a whole modernization process, so in a ADM-based process, all the other phases will also employ KDM. Therefore, this will make much easier the interoperability with the other modernization tools.

From the point of view of practical usage, this work can be seen as a starting point for motivating companies adopting KDM. Currently, there are several organization that have already adopted KDM in modernization processes (Ulrich and Newcomb, 2010), such as KDMAnalytics and ADA Software, however we envisage there are many other that could benefit from this standard so that their tools become interoperable.

As future works, we can mention the following: (*i*) Investigate the reusability of KDM solutions. We have developed in the last years several solutions that act over KDM, which makes these solutions platform and language-independent. For example, we have developed mining algorithms (Santibáñez et al., 2015; Santibanez et al., 2013), profiles (Santos et al., 2019a), refactorings (Durelli et al., 2017, 2014c,b). Other researchers have published solutions in the context of ADM (Pérez-Castillo et al., 2011; Bruneliere et al., 2010; Eclipse, 2017; Santos et al., 2019b; Durelli et al., 2014a). As the main goal of OMG is the interoperability of modernization tools, an important future work is the development of KDM-based modernization tools; (*ii*) Development of reverse engineering tools. The lack of parsers (discoveries) that generate KDM from other languages out of Java hinders a widespread adoption of KDM/ADM. Up to this moment, only Modisco provides good support for Java language. So, the lack of parsers for other languages, make the adoption difficult. (*iii*) The extension of Arch-KDM. In order to support indirect dependencies we will implement some algorithms to identify transitive dependencies that involve interfaces, inner classes and abstract classes. It is straightforward to implement it because the *AggregatedRelationship* class of KDM contains the "to" attribute. This attribute will be the new "from" attribute and we just will need to check that the type of the "to" for the new "from" must be of the type sought.

CRediT authorship contribution statement

André de S. Landi: Conceptualization, Methodology, Software, Investigation, Writing – original draft. **Daniel San Martín:** Writing – review & editing, Validation, Software, Visualization. **Bruno M. Santos:** Investigation, Writing – review & editing. **Rafael S. Durelli:** Supervision, Writing – review & editing. **Valter V. Camargo:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

Daniel San Martín would like to thank the National Agency for Research and Development (ANID) PFCHA/DOCTORADO BECAS CHILE/2016- 72170024. Valter Vieira de Camargo also would like to thank FAPESP (p.n. 2016/03104-0).

References

Abi-Antoun, M., Jonathan, A., 2008. Tool Support for the Static Extraction of Sound Hierarchical Representations of Runtime Object Graphs. ACM, pp. 743–744.

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

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

Avgeriou, P., Guelfi, N., 2005. Resolving architectural mismatches of COTS through architectural reconciliation. Lecture Notes in Comput. Sci. 3412, 248–257.

Bandara, V., Perera, I., 2019. Identifying software architecture erosion through code comments. pp. 62–69.

Bischofberger, W., Kühl, J., Löffler, S., 2004. Sotograph – a pragmatic approach to source code architecture conformance checking. In: Software Architecture. pp. 1–9.

Borah, B., Bhattacharyya, D.K., 2004. An improved sampling-based DBSCAN for large spatial databases. In: 2nd International Conference on Intelligent Sensing and Information Processing, (ISIP). pp. 92–96.

Bouckaert, R.R., Frank, E., Hall, M.A., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2010. WEKA–experiences with a Java open-source project. J. Mach. Learn. Res. 11 (5), 2533–2541.

Brunelière, H., Cabot, J., Dupé, G., Madiot, F., 2014. Modisco: A model driven reverse engineering framework. Inf. Softw. Technol. 56 (8), 1012–1032.

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

Çiraci, S., Sözer, H., Tekinerdogan, B., 2012. An approach for detecting inconsistencies between behavioral models of the software architecture and the code. In: 36th Annual Computer Software and Applications Conference, (COMPSAC). pp. 257–266.

Chagas, F., Durelli, R.S., Terra, R., Camargo, V.V., 2016. Kdm as the underlying metamodel in architecture-conformance checking. In: 30th Brazilian Symposium on Software Engineering, (SBES). pp. 103–112.

de Silva, L., Balasubramaniam, D., 2012. Controlling software architecture erosion: A survey. J. Syst. Softw. 85 (1), 132–151.

Deissenboeck, F., Heinemann, L., Hummel, B., Juergens, E., 2010. Flexible architecture conformance assessment with conqat. In: 32nd International Conference on Software Engineering, (ICSE). pp. 247–250.

Durelli, R.S., Santibáñez, D.S.M., Marinho, B., Honda, R., Delamaro, M.E., Anquetil, N., Camargo, V.V., 2014a. A mapping study on architecture-driven modernization. In: 15th International Conference on Information Reuse and Integration, (IRI). pp. 577–584.

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

Durelli, R., Santos, B., Honda, R., Delamaro, M.E., Camargo, V.V., 2014c. Kdm-RE: A model-driven refactoring tool for KDM. In: 5th Workshop on Software Visualization, Maintenance, and Evolution, (VEM). pp. 1–10.

Durelli, R.S., Viana, M.C., de S. Landi, A., Durelli, V.H.S., Delamaro, M.E., Camargo, V.V., 2017. Improving the structure of KDM instances via refactorings: An experimental study using KDM-RE. In: 31st Brazilian Symposium on Software Engineering, (SBES). pp. 174–183.

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

Eclipse, 2017. Modisco. http://eclipse.org/MoDisco/.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al., 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In: KDD. pp. 226–231.

da Fonseca, R.J.R.M., dos Santos Ponte Silva, P.J., da Silva, R.R., et al., 2007. Acordo inter-juízes: O caso do coeficiente kappa. Lab. Psicol. 160 (8), 81–90.

Garner, S., 1995. WEKA: The waikato environment for knowledge analysis. In: Proc New Zealand Computer Science Research Students Conference. pp. 57–64.

Hannemann, J., Murphy, G.C., Kiczales, G., 2005. Role-based refactoring of crosscutting concerns. In: 4th International Conference on Aspect-Oriented Software Development, (AOSD). pp. 135–146.

Herold, S., Mair, M., 2014. Recommending refactorings to re-establish architectural consistency. In: Software Architecture. pp. 390–397.

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

Hussain, S., 2013. Investigating architecture description languages (adls) a systematic literature review.

Ivkovic, I., Kontogiannis, K., 2006. A framework for software architecture refactoring using model transformations and semantic annotations. In: 10th Conference on Software Maintenance and Reengineering, (CSMR). pp. 10–144.

Knodel, J., Popescu, D., 2007. A comparison of static architecture compliance checking approaches. In: 2nd IEEE/IFIP Conference on Software Architecture, (WICSA). p. 12.

Koschke, R., 2018. Industrial experience on code clean-up using architectural conformance checking.

Landgrebe, T.C.W., Paclik, P., Duin, R.P.W., 2006. Precision-recall operating characteristic (p-ROC) curves in imprecise environments. In: 18th International Conference on Pattern Recognition (ICPR'06). pp. 123–127.

Landi, A.d.S., Chagas, F., Santos, B.M., Costa, R.S., Durelli, R., Terra, R., Camargo, V.V., 2017. Supporting the specification and serialization of planned architectures in architecture-driven modernization context. In: 41st Annual Computer Software and Applications Conference, (COMPSAC). pp. 327–336.
Maffort, C., Valente, M.T., Terra, R., Bigonha, M., Anquetil, N., Hora, A., 2016. Mining architectural violations from version history. Empir. Softw. Eng. 21 (3), 854–895.
Makhoul, J., Kubala, F., Schwartz, R., Weischedel, R., et al., 1999. Performance measures for information extraction. In: DARPA Broadcast News Workshop. pp. 249–252.
Matos, D.A.S., 2014. Confiabilidade e concordância entre juízes: aplicações na área educacional.
Murphy, G.C., Notkin, D., Sullivan, K.J., 2001. Software reflexion models: bridging the gap between design and implementation. IEEE Trans. Softw. Eng. 27 (4), 364–380.
OMG, 2012. Object management group (OMG) architecture-driven modernisation. http://www.omgwiki.org/admtf/doku.php?id=start.
OMG, 2016. Knowledge discovery meta-model (KDM). Available in http://www.omg.org/technology/kdm/, specification available in http://www.omg.org/spec/KDM/.
OMG, 2017. Architecture-driven modernization. http://adm.omg.org/.
Pérez-Castillo, R., de Guzmán, I.G.-R., Piattini, M., 2011. Knowledge discovery metamodel-ISO/IEC 19506: A standard to modernize legacy systems. Comput. Stand. Interfaces 150 (6), 519–532.
Perez-Castillo, R., Sanchez-Gonzalez, L., Piattini, M., Garcia, F., de Guzman, I.G.R., 2011. Obtaining thresholds for the effectiveness of business process mining. In: 5th International Symposium on Empirical Software Engineering and Measurement, (ESEM). pp. 453–462.
Pruijt, L., Köppe, C., van der Werf, J.M., Brinkkemper, S., 2017. The accuracy of dependency analysis in static architecture compliance checking. Softw. - Pract. Exp. 47 (2), 273–309.
Roncero, V.G., 2010. Classificação Semi-Supervisionada de Textos em Ambientes Distribuídos (Ph.D. thesis). Ph. D. dissertation, Universidade Federal do Rio de Janeiro.
San Martín, D., Camargo, V.V., 2021. A domain-specific language to specify planned architectures of adaptive systems. In: 15th Brazilian Symposium on Software Components, Architectures, and Reuse. In: SBCARS '21, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450384193, pp. 41–50.
Sangal, N., Jordan, E., Sinha, V., Jackson, D., 2005. Using dependency models to manage complex software architecture. In: 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA). pp. 167–176.
Santibáñez, D.S.M., Durelli, R.S., Camargo, V.V., 2015. A combined approach for concern identification in KDM models. In: 3rd Latin-American Workshop on Aspect-Oriented Software Development, (la-WASP). p. 10.
Santibanez, D., Durelli, R.S., Marinho, B., Camargo, V.V., 2013. CCKDM - a concern mining tool for assisting in the architecture-driven modernization process. In: 3rd Brazilian Conference on Software: Practice and Theory - Session Tool, (CBSoft). p. 1.
Santos, B.M., Landi, A.S., Santibanez, D.S., Durelli, R.S., Camargo, V.V., 2019a. Evaluating the extension mechanisms of the knowledge discovery metamodel for aspect-oriented modernizations. Journal of Systems and Software (ISSN: 0164-1212) 149, 285–304.
Santos, B.M., de Souza Landi, A., de Guzmán, I.G.-R., Piattini, M., Camargo, V.V., 2019b. Towards a Reference Architecture for ADM-Based Modernization Tools. In: Proceedings of the XXXIII Brazilian Symposium on Software Engineering. In: SBES 2019, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450376518, pp. 114–123.
Schröder, S., Riebisch, M., 2017. Architecture conformance checking with description logics. In: 11th European Conference on Software Architecture, (ECSA). pp. 166–172.
Sneed, H.M., 2005. Estimating the costs of a reengineering project. In: 12th Working Conference on Reverse Engineering (WCRE'05). pp. 9–12.
Stafford, A.L.W.J.A., 2001. Architecture-level dependence analysis for software systems. Int. J. Softw. Eng. Knowl. Eng. 11 (4), 431–452.
Terra, R., Valente, M.T., 2009. A dependency constraint language to manage object-oriented software architectures. Softw. Pract. Exp. 39 (12), 1073–1094.
Terra, R., Valente, M.T., Czarnecki, K., Bigonha, R.S., 2012. Recommending refactorings to reverse software architecture erosion. In: Proceedings of the European Conference on Software Maintenance and Reengineering. In: CSMR, vol. 150, pp. 335–340, (2).
Ulrich, W.M., Newcomb, P., 2010. Information Systems Transformation: Architecture-Driven Modernization Case Studies. Morgan Kaufmann Publishers Inc..
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers.
Yan, H., Garlan, D., Schmerl, B., Aldrich, J., Kazman, R., 2004. Discotect: a system for discovering architectures from running systems. In: 26th International Conference on Software Engineering, (ICSE). pp. 470–479.

André de Souza Landi is a systems analyst at INVILLIA working in a national project of UOL. He finished his master's at University of São Carlos - UFSCar/DC in 2018. Nowadays, he is researching about the topics of Software Architecture, Modularity, Model-Driven Engineering, Microservices and new techniques/framework for the Java language.

Daniel San Martín is a Ph.D. Student at Universidade Federal de São Carlos, Brazil and full-time professor at Exact Sciences Department at Universidad de los Lagos, Osorno, Chile. He received a B.S. degree in Engineering Science and Computer Engineering from Universidad Católica del Norte, Antofagasta, Chile and M.Sc. degree in Computer Science from Universidade Federal de São Carlos, SP, Brazil. He has experience as CISO, PM and Information Analyst in several public and private organizations. Currently, his research interests involves Adaptive Systems, Software Architecture and Model Driven Engineering.

Bruno Marinho Santos is graduated in Information Systems at Faculdade Integral Diferencial (FACID) in 2010 and he obtained his master degree in Computer Science in Software Engineering area at Federal University of São Carlos (UFSCar) in 2014. Nowadays, he is a Ph.D. student at UFSCar. He has experience in Computer Science area, with emphasis in Computation Systems, acting mainly in the following subjects: Aspect-Oriented Modernization, Architecture-Driven Modernization, Crosscutting Frameworks, Knowledge Discovery Metamodel, and metamodel extensions.

Warteruzannan is graduated in Computer Science at Federal University of Goiás and he obtained his master's degree in Computer Science at Federal University of São Carlos (UFSCar) in 2020. Currently, he is a Ph.D. student at UFSCar. He has experience in mobile/web development, machine learning, and software architecture.

Prof. Dr. Rafael S. Durelli is professor at Computer Science Department of Federal University of Lavras (UFLA) in Brazil. He finished his Ph.D. at University of São Paulo USP/ICMC in 2016. He is a member of PqES/DCC (Pesquisa em Engenharia de Software).

Prof. Dr. Valter Vieira de Camargo is an Associate Professor at Computing Department of the Federal University of São Carlos (UFSCar) in Brazil. He has co-authored around 130 research papers, covering the topics of Software Architecture, Software Modernization, Adaptive Systems, Modularity and Model-Driven Engineering. He finished his Ph.D. in 2006 and participated as a visiting researcher at University of Twente in 2013. He has also coordinated the AdvanSE (Advanced Research on Software Engineering) Group since 2009.