Mendivelso et al. Journal of Software Engineering Research and Development (2018) 6:16
https://doi.org/10.1186/s40411-018-0060-6

Journal of Software Engineering Research and Development

RESEARCH Open Access

Metric-centered and technology-independent architectural views for software comprehension

Luis F. Mendivelso[1,2], Kelly Garcés[1*] and Rubby Casallas[1]

* Correspondence: kj.garces971@uniandes.edu.co
[1]Department of Systems and Computing Engineering, School of Engineering, Universidad de los Andes, Cra 1 No 18A - 12, Bogotá, Colombia
Full list of author information is available at the end of the article

Abstract

The maintenance of applications is a crucial activity in the software industry. The high cost of this process is due to the effort invested on software comprehension since, in most of cases, there is no up-to-date abstraction or documentation to ease this task but the source code. The goal of many commercial and academic tools is to build software architectural views from the code. The main disadvantages of such tools are: i) they are dependent on the language/technology on top of which the application is built; and ii) they offer pre-defined views that are too difficult to adapt to meet particular software comprehension needs. In this paper, we present a Technology-independent approach which is flexible enough to allow developers to define metric-centered architectural views by using annotations. These views display in a single canvas architectural elements whose look and feel maps software metrics. Our work results from joint projects with industry partners with software modernization needs in different technologies: *Oracle Forms*, *Java EE*, and *Ruby on Rails*. We present how our proposal was applied in these projects and compare the results with those of the previously followed process.

Keywords: Software Comprehension, Software Visualization, Software Architecture, Model-Driven Reverse Engineering, Software Metrics

1 Extension Note

This paper is an expanded and revised version of the document entitled "Vistas Arquitectónicas Independientes de Tecnología para Comprensión de Software" (Mendivelso et al. 2017) presented at the Iberoamerican Conference on Software Engineering (CIbSE) 2017 in Buenos Aires (Argentina), held between May 22nd and 23rd of 2017. The paper extends the version that appeared in the Conference as follows: i) In the 'Motivation' section, we illustrate the need to have software metrics mapped to the views as a mean to leverage software comprehension. ii) In the 'View Generation Process' section, we explain our contribution in a deeper way compared to that of the initial version. Thus, we distinguish the assets involved in the model transformation chain that operationalizes our approach. We add a 'Metrics metamodel', a transformation to the chain, and tags to the original annotations. These tags describe how to modify the views style based on the metrics. iii) In the 'Evaluation' section, we present the applicability of our process to three case studies. We

spell out the cost of developing a solution under our approach, its benefits and its limitations. iv) In the 'Related work' section, we add relevant literature.

2 Introduction

According to (Minelli et al. 2014), the task of understanding a program corresponds to more than 50% of all the maintenance activity. The main reason for this is the absence of abstractions on the applications, generally with thousands of lines of code distributed in hundreds of files, which makes understanding more difficult.

In cooperation with industry partners we have carried out projects to tackle dif- ferent modernization challenges: 1) Migration from *Oracle Forms* to *Java* and *.Net (Garcés* et al. 2015*; Wikipedia* 2016*; Garcés* et al. 2018*)*; 2) Restructuring of *Java Enterprise Edition* (JEE) applications from mono- lithic architectures to microservices (Escobar et al. 2016); and 3) Maintenance of *Ruby on Rails (RoR)* applications developed by Agile practitioners (García and Garcés 2017). Literature (Anquetil and Laval 2011; Mancoridis et al. 1999) and our experi- ence in these projects have shown us that, besides to the related complexity with the size of the applications, the understanding of the programs is difficult for the following reasons: i) lack of documentation about the design; ii) lack of knowledge about architectural decisions taken in the original design because of developers' turnover; and iii) the degradation of previously made decisions, including additions and modifications done over time.

For every project, we reviewed tools that allow building abstractions of higher level, i.e., architectural views. For example, we found tools that obtain commodity views such as UML class (or package) diagrams from Java source code. We found that they are completely dependent on the language/technology, and that the views that they produce are predefined and do not necessarily correspond to particular understanding needs. Based on the experience on these projects, and taking inspi- ration from the general process defined by (Tilley 2009), we present an approach based on Model-Driven Reverse Engineering (MDRE) (Brunelière et al. 2014; Rugaber and Stirewalt 2004) that allows us to annotate a Platform Independent Metamodel (PIM) (referred to as *Architectural* model), and generate a specification of a graphic editor (which, for purposes of the evaluation, is Sirius[1] (Mendivelso et al. 2017)). Therefore, one can see in the editor different perspectives of the application according to the views defined in the specification; e.g., level of coupling between functional modules.

The contributions of our work are: i) A view generation process extensible to many source application technologies and views specification frameworks. Our approach can be applicable to many technologies by plugging new parsers to the workflow. Our proposal is view specification agnostic because the user specifies the views at architecture level by using annotations. The annotations are translated to the *Technology Independent Views Specification model* (TIVS) first, and then to the particular view specification framework (e.g., Sirius). ii) *Clustering algorithms* that group structural elements of the source applications, at an early step of the process. Thus, leveraging the last step which is *views render*. iii) *Annotations* that help users not only to specify the structural elements to be displayed, but also to represent particular software metrics via the elements style; that is, colour, size, and labels. For example, if there exists a relationship between two modules A and B, which is thicker than the relationship between B and C, then it means that the coupling

A-B is higher compared with that of B-C. Annotations refer to measures present in a *Metrics model*, which are calculated by a *Metrics transformation*.

When comparing our approach to the closest related work (http://themoosebook.org/book/index.html; Bergel et al. 2014), one finds that our approach resembles the competition's in two aspects: i) both are able to take, as input, artifacts that conform to different technologies; and ii) views style reflects software metrics. However, it distinguishes itself from the rest because: i) Whereas related work transforms source artifacts to a unique pivot, we have a pivot (referred to as *Architecture metamodel*) for each family of technologies. When having a unique pivot, there is the risk of finding no direct correspondences between the source technology and the intermediate representation. As a consequence, the user who develops the parser ends up establishing correspondences that suit her purposes, but that are not necessarily understandable by others. Therefore, our approximation aims at having a balance between reusability (among technologies of a same family) and expressiveness (keeping some semantics of the source tech- nologies). As an illustration, we have a pivot for 4GL languages that is useful to reverse engineering Oracle Forms, Visual Basic and Delphi. ii) In related work, the elements (and metrics) displayed in the view are limited to those explicit in the source structure. In contrast, in our approach, the user has the possibility of defining new structural elements (and corresponding metrics) needed for facilitating her software comprehension tasks. For example, in the Oracle Forms projects, our partners required the notion of functional modules, which is not present in Forms applications. iii) In related work, the views are rendered in editors under a fixed technology. In contrast, we have the TIVS model that can be mapped to different views specification frameworks (i.e., not only Sirius).

To validate our approach and tool, we have applied them in the concrete cases of *Oracle Forms*, *Java EE* and *Ruby on Rails* taking into account the needs that each project had. We were able to conclude that what was originally done in a specific way for a particular project could be largely generalized by our approach; thus, improving flexibility in the tasks of program understanding.

The remaining of the paper is structured as follows: Section 3 presents the moti- vation of our research based on the experience gained in the three modernization projects mentioned before. Section 4 elaborates on the MDE solution; i.e., the chain of model transformations to generate views for a particular source technology, and how the chain is parameterized with annotations. Section 5 presents an evaluation. Section 6 extends related work. Section 7 summarizes conclusions and perspectives for the future.

3 Motivation

3.1 Migration from oracle forms to java and net

Oracle Forms appeared at the end of the 1980s as a programming language and development tool to create Client/Server applications that interact with Oracle databases. *Forms* minimizes the need of developers to program common operations like transactions management and coding of CRUD operations. The applications can include *PL/SQL* code that allows programmers to enrich the functionality be- yond CRUD. Forms and Tables are two of the main concepts supporting *Forms* applications.

The challenges in this project were related to knowing, given a form to be mi- grated, which are the tables and forms related to it? which functional module does contain it?

and, how is that module related to others? Answering these questions was important to delimit the migration scope. Similarly, the project was challeng- ing since it lacked: documentation, availability of those who initially developed the legacy system, directory hierarchy that organizes the forms, significant naming con- ventions, and relationships embedded in the *PL/SQL* code, which is scattered along the elements of the forms. Another challenge is the size of the applications. In this project, the number of forms ranges from 83 to 178, referenced tables from 101 to 200, blocks from 361 to 765 and triggers from 2140 to 4406.

When reviewing the state of the art in terms of views for *Oracle Forms*, we found tools that showed the graphic interface design, or a navigation tree of the application structural elements. These tools are Oracle2Java[2]], Evo,[3] Jheadstart,[4] Pitss,[5] Ormit.[6] Since this was not enough to meet the challenges, three views were proposed (Garcés et al. 2015): the *Functional modules* view where each module group forms. The modules are represented with circles which size changes according to the amount of forms in it. This view shows the relationships between the modules that result from synthesizing dependency relationships between forms. Additionally, the view hides the content of each module since it is the access point to another more detailed view: the *Forms* view. This view shows the forms of a selected module. In this view, the size of the elements depends on a metric which is the number of elements that compose the form. Finally, the *Forms and Tables* view shows the different kinds of relationships between *Oracle Forms* elements: between forms and tables (e.g., simple and master/detail relations) and between forms (e.g., call relation). Fig. 6b illustrates the Forms and Tables view.

3.2 Restructuring monolithic JEE applications with microservices

JEE is commonly used in business applications and it proposes a three-layer architecture: presentation, logic, and persistence. The project focused on the logic and persistence layers, where there are *Enterprise Java Beans* (EJBs).

The challenges in this project were understanding an existent architecture (in particular, the coupling level of the logic and persistence *EJBs*), and proposing a partition of the *EJBs* in microservices. For the latter, it was defined as a principle that the highly coupled *EJBs* were grouped together and that the loosely coupled ones were isolated and presented as independent microservices. It was important to provide a list of relations (i.e., invocations) between the microservices to decide whether or not a set of given *EJBs* may be split. At last, it was key to understand how the microservices operate the database tables in execution time to verify that each microservice accesses different tables and, thus, respect the isolation principle. Satisfying these challenges was important to ease architectural decision making about how to evolve monolithic applications to microservices. With respect to the challenge of size, the studied application had 74566 LoC, 624 classes, and 35993 methods.

We found many tools that produce UML diagrams from Java code; for example: Enterprise Architect,[7] Modelio,[8] IBM Relational,[9] StarUML,[10] among others. However, the proposed diagrams are very generic for the migration purpose. We proposed four views that respond to the particular needs of the project: The *EJBs clusters* view in which every cluster groups *EJBs* of logic and/or persistence. The clusters were represented with circles which dimension is a function of the number of contained *EJBs*. To rank the number of lines of code (LOC) of a cluster, we assigned a color to each circle

by using Saffir-Simpson scale (Wikipedia 2017). The *Microservices* view groups the clusters of the previous view. The *Microservices and tables* view shows the diverse types of relationships that can be established between microservices and database tables: writing, reading, updating and deletion relations. Finally, there is a view that shows the dependencies between the logic *EJBs* that are in the microservices. Fig. 6d illustrates the EJBs clusters view.

3.3 Maintenance of Ruby on rails applications

Ruby on Rails is an agile development framework for web applications that follows the architectural pattern Model-View-Controller.

The challenge in this project was knowing which are the *Rails* models (i.e., per- sistence entities, services and utilities)? how are these related to one another? and which is the type of a given relation? This was important in the agile development context to determine the viability of a new user story and estimate developers' effort to implemented it. Finally, *Ruby* is a dynamically typed language that has differences between its versions. Regarding the challenge of the application size, in this project, the applications consisted of 15 to 51 Rails models.

Some of the tools we found to face this challenge were: *Railroady*[11] and *MySQL Workbench*, [12] which offer views that are hard to navigate and lack detailed in- formation about which are the attributes/methods of a given *Rails* model. That motivated a new view that shows all the *Rails* models (including their attributes and methods) and the relationships between them (i.e., simple, aggregation and composition). In contrast to the rest of projects, *Ruby on Rails* did not require any view customization to reflect metrics. Fig. 6c illustrates the Rails view.

3.4 Analysis of common features

We have observed the following common needs in the previous projects: It is required to have views that: i) show *coarse-grained elements* (i.e., containers) that group fine-grained elements; ii) deploy the detail of every coarse-grained element;

iii) combine coarse and *fine-grained elements* (i.e., nodes) into one container; iv) show *relationships* between elements; v) be navigable from a high to a low level of detail; and vi) have visual aids (e.g., labels, colors, sizes) to reflect a software metric. Table 1 summaries the views needed in each project and how technology/architec- ture concepts map to the aforementioned view concepts.

4 Views generation process

4.1 Overview

Even though the projects worked on different technologies, at the end, as we pointed out in Section 3.4 the architectural visualization needs are similar. Thus, we decided to study the possibility of generating metric-centered and technology-independent architectural views. We must emphasize that our aim is creating the most common views observed both in the literature and in our own experiences. Such views serve as a starting point but may require customization to meet all the particular needs. For each new technology we have to develop some assets (referred to as the *Domain Engineering* process) that are then reused for every application that conforms to such a technology

Table 1 Mapping between technology concepts and view concepts

Project	View	Technology concept	View concept
JEE	Microservices View	Microservice	Container
		Microservice Relationship	Relationship
	EJB Clusters View	EJB Cluster	Container
		EJB Cluster Relationship	Relationship
	Microservices and Tables View	Microservice	Container
		Table	Node
		Writing Relationship	Relationship
		Reading Relationship	Relationship
		Updating Relationship	Relationship
		Deletion Relationship	Relationship
	EJB Dependencies Views	Class	Container
		Interface	Container
		Attribute	Node
		Method	Node
		Generalization	Relationship
		Realization	Relationship
		Composition	Relationship
		Dependency	Relationship
Ruby on Rails	Rails View	Package	Container
		Rails Class	Container
		Attribute	Node
		Method	Node
		Rails Relationship	Relationship
Oracle Forms	Functional Modules View	Module	Container
		Module Relationship	Relationship
	Forms View	Form	Node
	Forms and Table Views	Form	Node
		Table	Node
		Single Table Relationship	Relationship
		Master-Detail Relationship	Relationship
		PL/SQL Relationship	Relationship
		Form Call Relationship	Relationship

(this is called the *Application Engineering* process). These terms are originally coined in (Czarnecki and Eisenecker 2000). Figure 1 (which uses a data flow notation) shows the two processes. The *application engineering* level is for the final users (such as architects, developers, testers, etc. who carry out software comprehension tasks) to generate metric-centered views for a particular application. The final users provides the application data to analyze and automatically, the process generates the views of the application. This process consists of 4 steps (represented as circles): i) data injection to obtain a memory representation of the input software artifacts; ii) queries on the representation of step 1 that allow us to find the architectural elements of the language/technology; iii) computation of metrics related to such elements; and iv) rendering of architectural elements and their associated metrics in an editor.

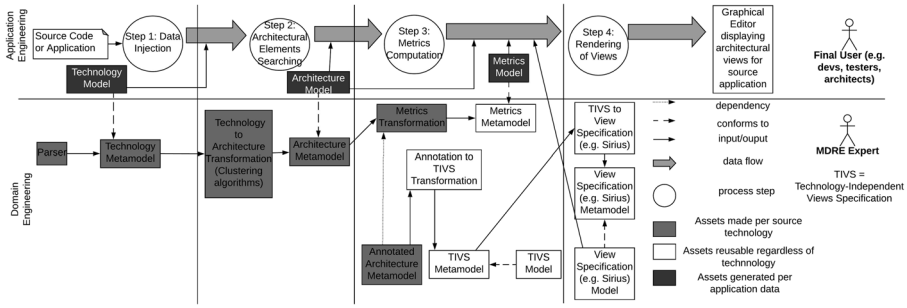

Fig. 1 Process at application/domain engineering level

The *domain engineering* level is for an MDRE expert who has to provide part of an infrastructure and reuses other part. Light gray squares represent the assets that the MDRE expert has to define per each new technology or programming language of interest. White squares represent assets at the heart of our approach that can be reused regardless of the source technology or language. Dark gray squares represent the assets generated per each source application to be analyzed. The assets consist of parsers, models, metamodels and transformations. Next sections describe these assets in detail. We use the Oracle Forms project as illustrating example.

4.2 Step 1: Parsers

For step 1, depending on the technology, the MDRE expert has to produce a model that conforms to the language/technology metamodel from the source data. This task means to develop or reuse one or more parsers depending on the available input data. In our experimentation, in the case of *Oracle Forms*, we developed a single parser in Java that produces a model conforming to a metamodel of *Oracle Forms*. For *Ruby* code, we implemented two Xtext grammars[13] which in turn generated the parsers and the meta-models for SQL and *Rails*. For *JEE*, we used Modisco,[14] which gives as a result a model with a full abstract syntax tree of the source code. Such a model conforms to the standard metamodel so-called KDM.[15]

Besides the source code, we had dynamic data, i.e., logs that saved a trace of the SQL operations executed. This was useful to complete the model with relationships between the database tables and the microservices. A specific parser was built for that task.

4.3 Step 2: architecture metamodel and technology2architecture transformation

The MDRE expert has to design an architecture metamodel and a Technol- ogy2Architecture transformation for this step. The architecture metamodel rep- resents the structural elements that final users want to have in the views. The Technology2Architecture transformation produces an architecture model from the step 1 model.

Fig. 2 shows the architecture metamodel for the Oracle Forms industry case. It provides a way to specify the main concepts needed to represent a typical clien- t/server 4GL application but tries to do so in a more platform independent repre- sentation (in the MDA sense of the word). This would enable in the future to reuse most of the metamodel in the domain engineering process of other similar languages (like Delphi or Visual Basic).

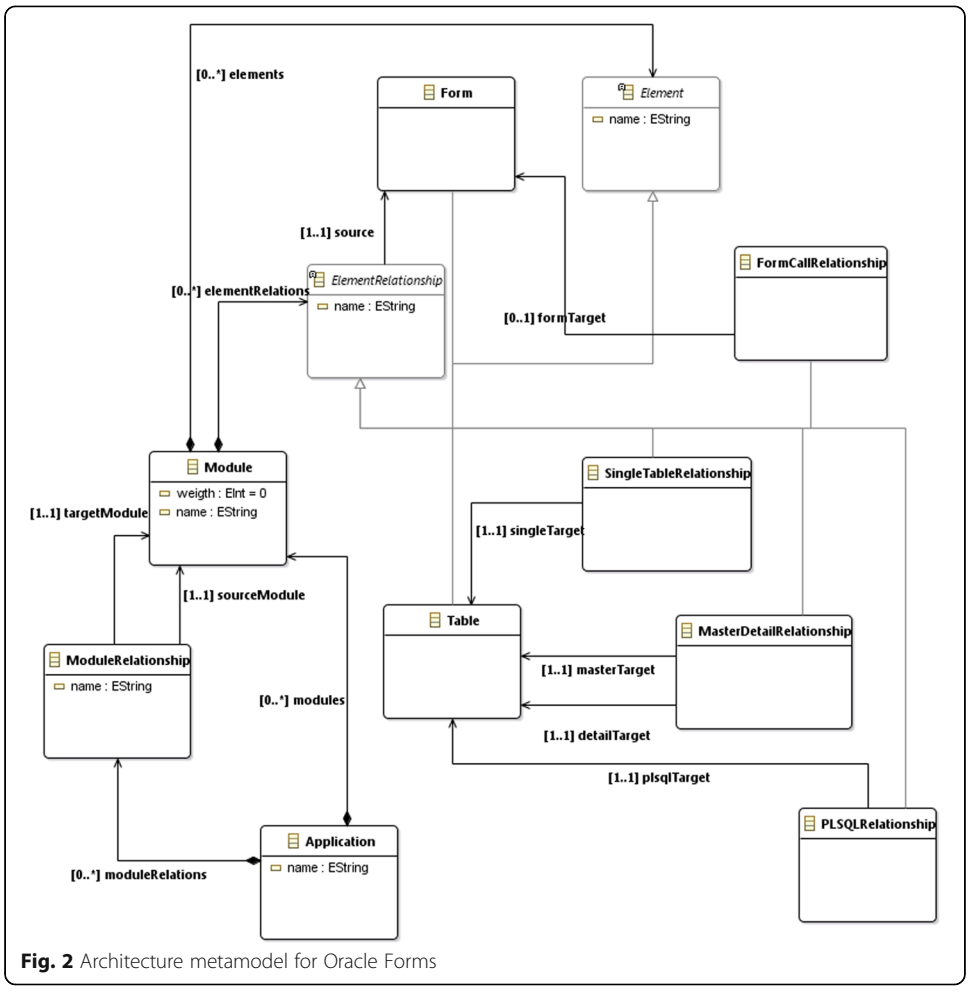

Fig. 2 Architecture metamodel for Oracle Forms

Some of the main metamodel concepts are: i) 'Module' that contains 'Form' and 'Table'; ii) 'ModuleRelationship' that represents the relations between the modules; and iii) A 'SingleTableRelationship' that describes a simple relation between a form and a table. It is worth highlighting that some of these concepts have a direct corre- spondence with the concepts of the technology models (i.e., Form and Table). How- ever, others are calculated from the information contained in those models by using clustering algorithms in the Technology2Architecture transformation. A clustering algorithm arranges fine-grained elements in coarse-grained elements by evaluating their attributes and relationships. Previous works in software comprehension have used clustering algorithms, see for example (Anquetil and Laval 2011; Mancoridis et al. 1999).

Clustering algorithms were needed in the three industry cases. However, their use should be assessed on a case-by-case basis because it depends on the final users' requirements and the software structure. In the next paragraphs, we spell out the requirements and software restrictions that motivated the use of clustering in the industry cases, as well as a brief description of the algorithms.

In the Oracle Forms case, the fact of knowing the module containing the form sub- ject matter of modernization helps developers to delimit the modernization scope. However, it is not always easy to get the modules because legacy software organi- zation is often quite poor. To cope with this, we have implemented two clustering algorithms that arrange the

forms, tables and relationships into modules: *Menu- based clustering algorithm* and *Table betweenness clustering algorithm (Garcés* et al. 2015*)*. 'Module' is the result of these algorithms that group in a module the forms that they have in common, or that are called from a same menu of the graphic interface, or that depend on the same tables. In addition, 'ModuleRelationship' comes out the calls between forms that are contained in two different modules (the calls are derived from CALL/OPEN queries embedded in the PL/SQL code).

In the JEE case, developers need suggested options about how to split the legacy applications in microservices. This distribution is not trivial since stakeholders likely face legacy code with the following problems: 1) lack of design and documentation; 2) absent reasoning on the original design and architecture decisions; 3) undermin- ing of the original design decisions as many additions and alterations have been made. To deal with this, we implemented an algorithm that groups together the logic *EJBs* that access to the same persistence entities. In addition, we developed another algorithm that groups *clusters* in microservices, taking into account the per- centage of *EJBs* in common. These algorithms are referred to as *EJB Clustering*, and *Microservice Clustering* in previous work (Escobar et al. 2016).

In the Ruby on Rails case, developers need to have a wide vision of the application before taking (re)design decisions. This vision encompasses diagrams for the logic and persistence layers of Ruby on Rails applications. This is challenging because, in applications built on top of dynamically typed languages (like Ruby), it is not possible to have all the information of methods and attributes that will be available in execution; besides that, the types of the attributes are unknown and thus it is impossible to determine which classes have aggregation/composition relationships. To address this, we have implemented a clustering algorithm that groups *Rails* models in packages based on namespaces and discovered correlation between classes and database tables (García and Garcés 2017).

This helps illustrate that an intermediate representation (i.e., the architecture model) is necessary, where information is synthesized according to interest criteria through the clustering algorithms, and that occurs before views specification; otherwise, the implementation of the latter would be very complex.

4.4 Step 3: Metrics transformation and annotations

The MDRE expert has to carry out the following three tasks in order to guarantee the success of step 3:

1) *Development of a metrics transformation* that queries a model conforming to the architecture metamodel and creates measures and measurements in an output model. This output model is conforming to the metrics metamodel.
2) *Annotation of the architecture metamodel* to indicate which structural ele- ments of the language/technology will be painted, and their appearance. This task has a strong dependency with the previous one because the measures referenced in the annotations are influenced by the measures introduced by the metrics transformation. We have chosen the annotations as the expres- sion mechanism because they are metadata that do not affect the metamodel nor the related models. Our tool validates that the annotations have been correctly applied to the architecture metamodel.

3) *Generation of a Technology-independent Views Specification (TIVS) model* which is an intermediate representation of the view specification that allows us to be independent from the concrete graphic framework on top of which the view editor is built. This model conforms to the TIVS metamodel (see Section 4.4.2) and is automatically generated from the annotated architecture metamodel by using a transformation (referred to as Annotation2TIVS in Fig. 1).
The two first tasks are manually carried out by the expert and the latter one is automated through a transformation.

4.4.1 Metrics transformation

The metrics transformation is responsible for: i) creating measures in the metrics model; ii) querying the architecture model to obtain measurements for the established measures; and iii) associating architecture elements to measurements in order to have a trace indicating where do the measurements come from. The transforma- tion output is a model conforming to the metamodel shown in Fig. 3. To design this metamodel, we took inspiration from the Structured Metrics Metamodel (SMM) which is an OMG specification (OMG n.d.). The main concepts of SMM are: *Measure, Mea- surement* and *Observation*. Measure defines a type of software metric that can be applied to one or several architectural elements. Measurement contains the concrete value of a metric for a particular architectural element. Observation saves a trace of which tool is used to compute the measurements, when, and who is the responsible for the observation.

In our Metrics metamodel, we keep the Measure and Measurement concepts and leave aside the Observation concept. The latter adds no value to our approach, instead it impacts the metrics model size and the performance of the views render.

In addition, the SMM includes a set of software measures (that extends the con- cept Measure) but makes no claims about its standardization. Thus, We decided to put aside these subclasses too and prefer to keep our Metrics metamodel as minimal as possible. In fact, it is the MDRE expert the one who adds the measures of her interest at model level.

- *MetricSystem*: Describes the root concept.
- *Measure*: Corresponds to the Measure concept from the SMM.
- *Measurement*: Corresponds to the Measurement concept from the SMM. It is worth noting that the measurement has a reference whose opposite is kind of Object. This allows us to reference any kind of element being present in the architecture model.
  Listing 1 shows an excerpt of the Metrics transformation developed for the Oracle Forms case. This excerpt is devoted to associate the measure 'Total number of Tables' to the modules. To this end, the transformation: i) creates the measure (lines 1-2); ii) obtains the collection of modules contained in the application (lines 4-6); and iii) computes the number of tables for each module and assigns this value to the measurement (lines 6-8). It is worth noting that the measurement is linked to the measure (line 9) and the architectural element (line 10) from which the value is computed.

Listing 1 Example of a Software Metrics Transformation for Oracle Forms

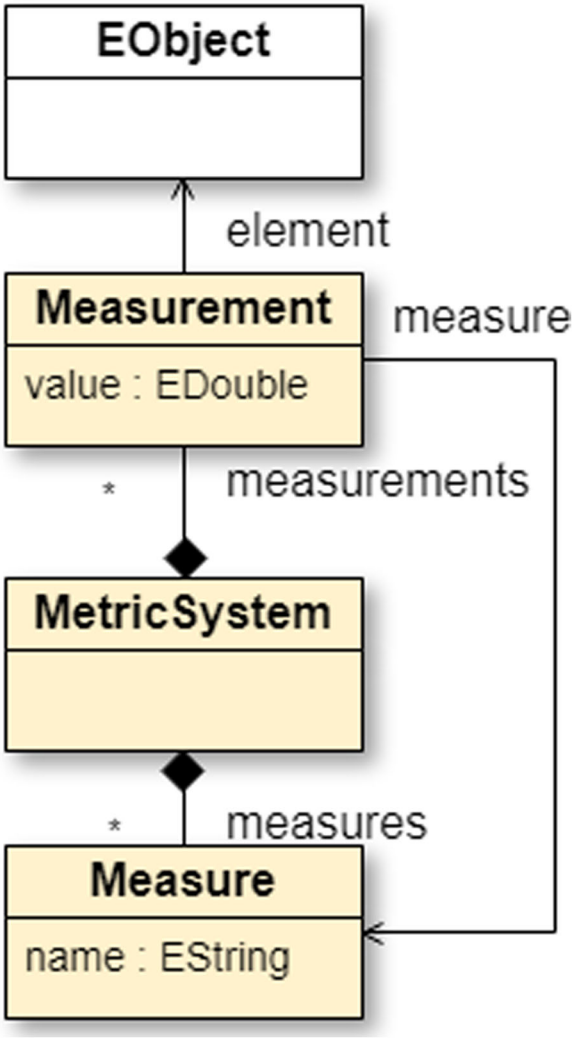

Fig. 3 Metrics metamodel

```
1  var measure = new Measure ( ) ;
2  measure . name = ' Total number o f Tables ' ; 3

4var mai n App l ica t ion = OracleFormsMM ! A p p l i c a t i o n . a l l I n s t a n
c e s ( ) . f i r s t ( ) ; 5

6  for ( module in mai n App l ica t ion. . modules ){
7  var measurement = new Measurement () ;
8  measurement. v a l u e = module. e l e m e n t s. s e l e c t ( e | eocl Is Type Of
   (Table ) ). . s i z e () ;
9  measurement. Measure = measure;
10 measurement. Element = module; 11}

12...
```

4.4.2 Annotations and Technology-independent Views Specification (TIVS) metamodel

The proposed annotations are aligned with the TIVS metamodel. Therefore, before explaining the annotations, we present such a metamodel. We decided to repre- sent metrics in a metamodel separated from the TIVS metamodel to individualize concerns. Figure 4 shows the TIVS concepts and a basic description follows. The application of these in the projects is introduced in the next chapter.

- *System*: Describes the root concept.
- *Cluster*: Represents a set of elements that have been grouped according to a given criteria.
- *ClusteredElement*: Represents the content elements inside of a *cluster*.
- *Relationship*: Describes an association between Clusters or ClusterElements.
- *Element*: Abstract concept from which Cluster and ClusteredElement extends. It allows that a relation can have as source/target two Clusters or one Cluster and one ClusteredElement. Additionally, it allows that a Cluster can contain others Clusters or ClusteredElements.

Annotations: In this section, we describe the annotations:

- *@Cluster*: It is used in a non-abstract metaclass that represents a Cluster.
- *@ClusteredElement*: It is used in a non-abstract metaclass that represents the ClusteredElement of a metaclass annotated with @Cluster. The annotation cannot

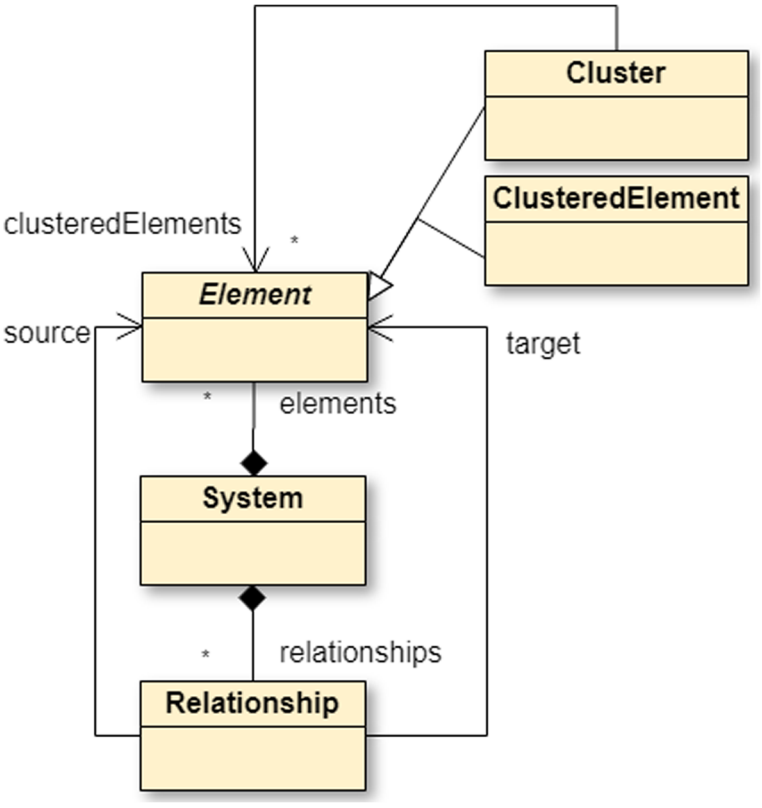

Fig. 4 Technology-independent Views Specification (TIVS) metamodel

be placed on superclasses but directly on the terminal class that represents the actual element to be painted.

- *@ClusteredElementCollection*: It is used inside a metaclass annotated with
- @Cluster; in particular, it must be placed on composition references of that metaclass whose type is that of classes previously annotated with @ClusteredElement or @Cluster.
- *@Relationship*: It is defined on a metaclass that represents in itself a rela- tion between two metaclasses A and B. It is mandatory that both A and B are annotated either with Cluster or with ClusterElement. The annotation
- @Relationship is accompanied by the following two additional annotations: i) *@source*: It is used on a reference of the annotated metaclass with @Relation- ship that points to the source of a relationship; ii) *@target*: It is defined on a reference of the metaclass annotated with @Relationship and is an annotation that points to the target of a relationship. In addition, it is possible to specify the line style, where the possible values are: solid, dash, dot or dash-dot. If final user requires a different style, then MDRE expert has to define it in the views specification (in this case, Sirius).
- All annotations with the exception of *@ClusteredElementCollection* can in- clude three graphical properties: color, size, and label. Developers have to map these properties to measures from the metrics model. That is, the ap- pearance of the displayed architecture elements is changed as a function of the measurements computed in step 3.

  The List 2 shows fragments of the architecture metamodel for Oracle Forms using the notation editor *OCLinEcore.*[16] It also shows how the annotations have been applied.

  In the List 2, we have annotated the class 'Module' with 'Cluster' since it is the only desirable grouper, the classes 'Table' and 'Form' with 'ClusteredElement' since both are contained in the 'Module' to visualize, the relationship 'elements' between 'Module' and 'Element' with 'ClusteredElementCollection', which allows the access to the tables and forms. In addition, the class 'SingleTableRelationship' was annotated with 'Relationship' to obtain the link between forms and tables. Finally, we include two graphical properties in the 'Cluster' annotation and one in the 'Relationship' annotation. The first two are size and color; these depend on the measures "Total numbers of Forms" and "Total number of Tables" respectively. The last property is label, and it is mapped to the "Lines Of Code" measure. The name of measures used in the annotations must match the measures defined in the metrics model.

Listing 2 OracleForms Architecture Metamodel Annotated

```
1 class Application{...
2 property modules: Module [* | 1] {composes}; 3}
4 @ Cluster (size =' Total numbers o f Forms', color =' Total number o f
  Tables')
5 class Module {...
6 @ ClusteredElementCollection
7 property elements: Element [* | 1] {composes};
```

```
8  property elementRelations: ElementRelationship [* |
   1] {composes};

9}

10 @ Clustered Element ()
11 class Form extends Element {. ..}
12 @ Clustered Element ()
13 class Table extends Element {. ..}
14 abstractclass ElementRelationship {. . .
15 @ source
16 property source: Form [1 ] ;

17}

18 @Relationship (label =' Lines Of Code', style =' Solid')
19 class SingleTableRelationship extends ElementRela
   tionship {. . .
20 @ target
21 property singleTarget: Table 1;
22 }
```

4.5 Step 4: Views specification model

In this step, the MDRE expert uses the TIVS2ViewSpecification transformation, which is an asset useful regardless of the source technology. This transformation takes as input the TIVS model and produces as output a views specification model.

A *views specification* model is used for generating the graphic editor code that will paint the elements in the views. The specification covers three aspects: i) a *graphic model* that specifies the visual forms that will be used in the editor (e.g., a gray rectangle, a continuous line); ii) a *model with correspondences* between the visual forms and the metamodel elements (e.g., instances of the class X are represented by the gray rectangle specified in the graphic model); iii) *actions* that allow the user to interact with the views (e.g., click on a menu to move from one view to another); and iv) *conditional style for graphical model* that modifies some characteristics such as colors, sizes or labels, according to a specific data (e.g., the color of a rectangle).

In our approach, the *views specification* model conforms to the Sirius meta- model and is automatically generated from the TIVS model by using the TIVS2ViewSpecification transformation (see Fig. 1). Some of the correspon- dences between these two last models are: an annotated class with *@Cluster* is mapped to instances of the classes *Viewpoint, DiagramDescription, NodeDescrip- tion and ContainerDescription*. An annotated class with *@Relationship* is mapped to *EdgeDescription*. Please refer to Sirius documentation for more details about the underlying metamodel.

In the application domain phase, the architecture model and the metrics model are taken as input by the view specification model to display, in the graphical editor, the

views for the source application. The resulting views depend on the annotated architecture metamodel and can be categorized as follows:

- *Clusters views*: shows a set of clusters and the relationships between them. Examples of these kinds of relationships are: functional models (*Oracle Forms*), *EJBs* clusters and microservices (JEE).
- *ClusteredElements for Cluster View*: shows in detail the content elements in a selected *Cluster*. As an example, there exists the *Forms and tables* view (*Oracle Forms*).
- *ClusteredElements Global View*: shows all ClusteredElements of the system, of a same kind, without differentiating to what *Cluster* they belong. An illus- tration of this is the *Forms* view (Oracle Forms).
- *Cluster and ClusteredElements (or combined) view*: shows Cluster and ClusteredElements together to discriminate what elements belong to what cluster, and determine the relationships among these elements. An example of this view is the (*Ruby on Rails*) model view.
  As previously mentioned the TIVS model is an intermediate asset that allows us to separate our solution from the graphic framework, e.g., Sirius. Our tool takes this model as input and generates as output the Sirius specification model. Figure. 5 gives a flavor of how the Sirius specification looks like. This excerpt of the specification was generated from the class 'Module' that was annotated with '@Cluster'. A description of the elements follows:
- A Viewpoint element (called 'Viewpoints for Cluster Module') that contains a 'DiagramDescription' for each kind of view of our approach. 'Clusters for Module' is one of them.
- 'Clusters for Module' contains a 'NodeDescription' (i.e., 'Cluster-Module') that defines a representation for the model instances conforming to 'Module'. In the case of 'Total number of Tables', the color of the 'NodeDescription' has to change based on the number of tables contained in the 'Module'. Sirius allows this via a 'ConditionalStyle' that consists of a set of 'PredicateEx- pressions'. In our case, each 'PredicateExpression' is a query that obtains a measurement value from the metrics model and assigns a corresponding color. A query example is shown in Listing 3. This query seeks a measurement that meets two conditions: i) refer to the measure 'Total numbers of Tables'; and ii) point to the current module. If the measurement value is less than or equal to threshold, then a particular color is assigned.

Listing 3 'PredicateExpression' for Oracle Forms case.

```
1(Metric System. a l l I n s t a n c e s ()– > collect (measurements)– > select (m|m. measure. name =' Total numbers o f Tables' and m. element = self)– > a s Ordered Set ()
– > f i r s t (). v a l u e). f l o o r () < = 1 5;
```

5 Evaluation

5.1 Scope

In this section, we present an evaluation that focuses on showing the flexibility of our solution. By 'flexibility' we mean the ease with which our approach can be modified for

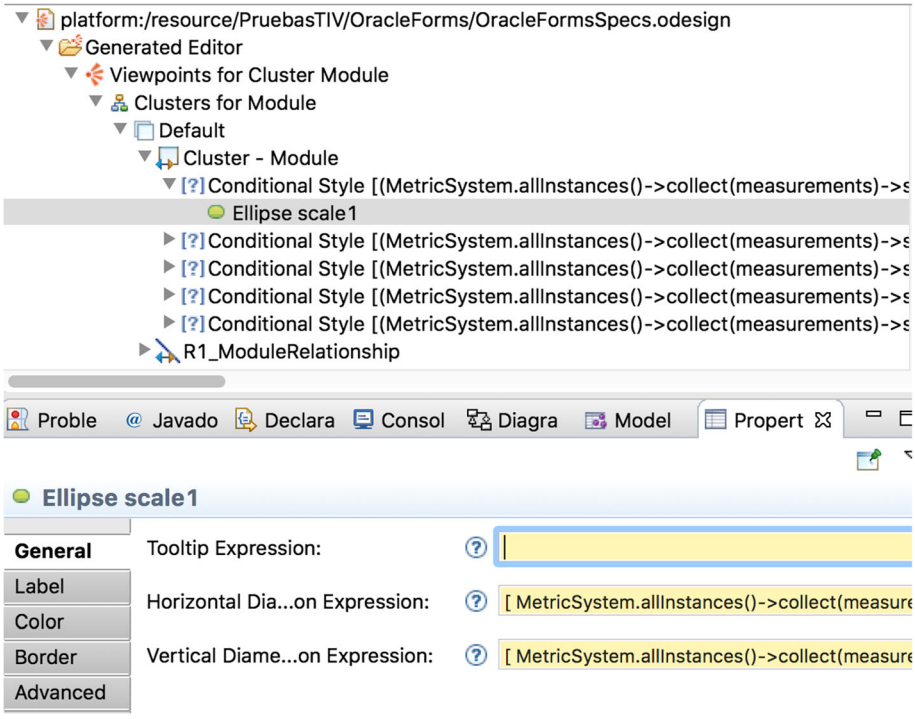

Fig. 5 Excerpt of the view specification model generated for the Oracle Forms case

use in various source technologies. We chose the technologies of the industrial cases (see Section 3) which are quite different from one another: Oracle Forms is a four-generation programming language, Java and Ruby are Object Oriented Programming languages that range from statically to dynamically typed. We present the results obtained in the three aforementioned industrial cases and finally a discussion is presented.

6 Results

The size of the generated views specifications ranges from 44 (for *Forms*) to 134 elements (for *JEE*). This difference is due to the fact that for each class annotated with @Cluster, the tool creates by default the four types of view and a series of scaffolding elements required by the technology of graphic editors, i.e., Sirius. The Figure 6a shows the forms and tables views of *Oracle Forms* when the generated specification is used. In these one, the gray rectangles represent the forms and tables, and the gray lines the relations between them. We included this figure to illustrate the default appearance of the views when the MDRE expert does not use the annotation elements related to size and color.

In the case of Oracle Forms, 3 out of 3 expected views were generated. For the Forms and Tables view of Oracle forms, an additional EgdeDescription for 'Mas- terDetailRe- lationship' was included because this one, as opposed to the other ones, has a source and two targets; this is, a form and the master and detail tables. Our approximation is limited to relations with a unique source and target. A workaround would be to annotate a same class from the metamodel with *@Relationship* several times, indicating for each annotation a different pair source-target.

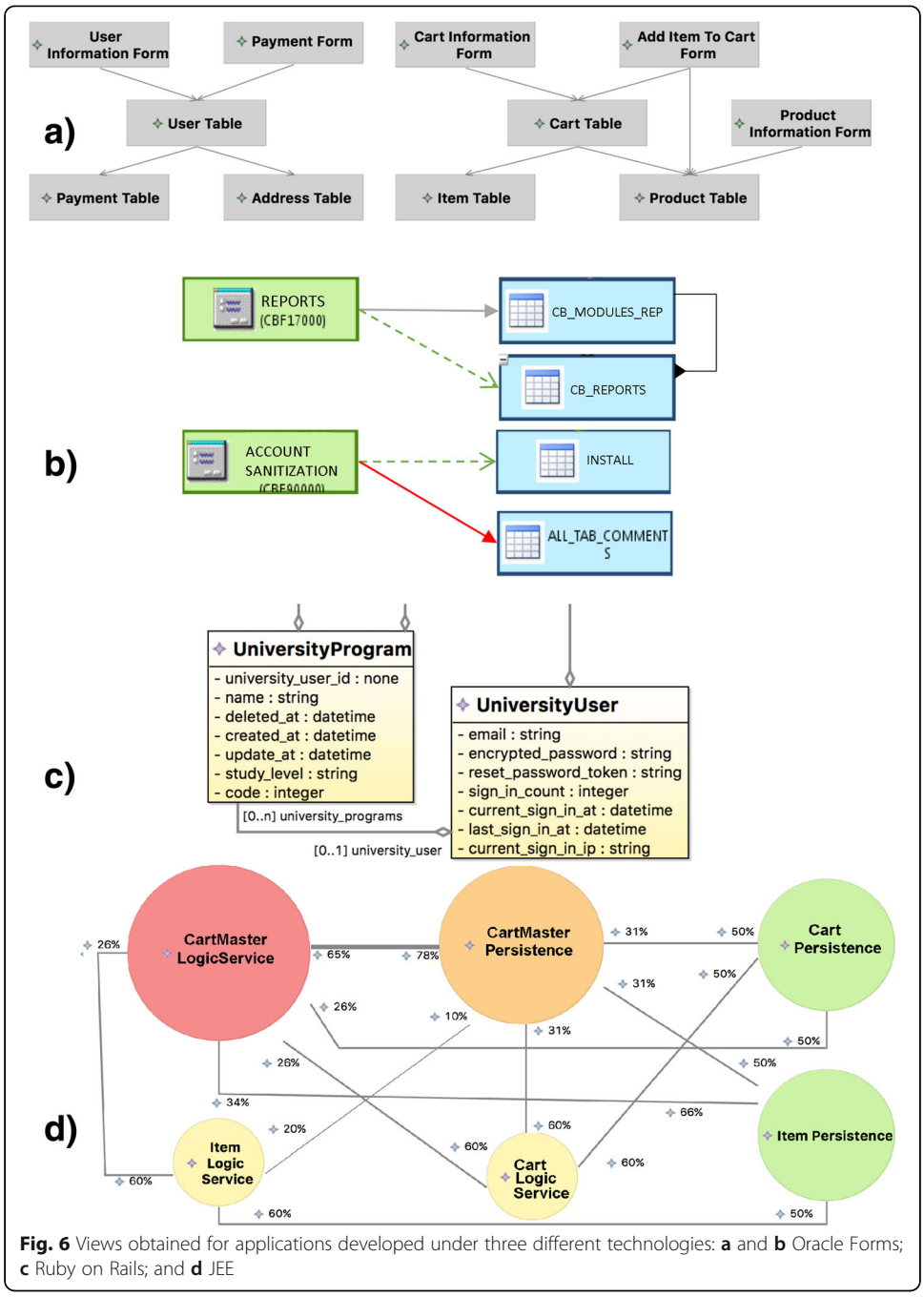

Fig. 6 Views obtained for applications developed under three different technologies: **a** and **b** Oracle Forms; **c** Ruby on Rails; and **d** JEE

In the JEE case, 2 of the 4 expected views were produced. The view about depen- dencies among *EJBs* had to be done by hand, because in the architecture metamodel there is not a class that represents in a synthetic way the relationships between the EJBs; what is described is that the dependency relationships have an invoking method and one invoked. As a consequence, we had to write a query that navigates dependencies between methods to, from there, derive the calls between EJBs. Simi- larly, the specification for the Microservices and Tables view was made by hand since the instances of 'Table' are contained within 'Architecture' instead of being inside 'Microservice'. The proposed decomposition views assume that

in the architecture model the container and content elements share a direct composition reference.

In the case of Ruby on Rails, the expected view that combines Rails models with their attributes, methods and relations was generated. In this case, there was no need for the annotation elements related to graphical properties. The reason was the simplicity of the view; Sirius' default configuration was sufficient enough to adjust the size of *Rails* models to the number of displayed attributes and methods.

Given the generated specifications, we have obtained the views (see Fig. 6) for applications written in the different technologies. In *OracleForms*, the specification was tested in 4 applications of the insurance and banking sector with sizes between 100 to 200 tables, and 83 to 178 forms. In *JEE*, in 3 applications of the academic, commerce and government domains, with sizes of up to 624 classes. In *Ruby on Rails*, in 3 applications designed for professional and educational services with a size of between 45 and 153 Rails models. Fig. 6d demonstrates that the size and color of EJBs clusters (represented as ellipsis) are different among them. In the architecture metamodel for this case, the MDRE expert defined that the size and color of the EJBs clusters depend on the number of EJBs and LOC contained respectively.

7 Discussion

7.1 Cost of developing a solution

The cost of a solution for a particular technology/language depends on the effort of developing the assets (represented as light gray squares in Fig. 1) required in each step of the views generation process. The assets of steps 1-2 are mandatory regardless of whether our approach is being used or not (Tilley 2009). Instead, the adoption of our approach results in savings in steps 3-4 because the MDRE expert designs the views by annotating the architecture metamodel without caring about the scaf- folding needed in the particular views specification framework (e.g., Sirius). Let us explain the scaffolding needed in a Sirius specification (i.e., *.odesign* file).

The MDRE expert has to establish correspondences between the architecture metamodel and Sirius concepts: i) The 'Module' class contains the forms and tables that one wants to see and therefore the module should be mapped to ContainerDe- scription ii) The 'Table' and 'Form' classes are fine-grained elements so that they correspond to NodeDescription, and iii) The 'SingleTableRelationship' class has to be visualized as a link whose source points to a form and target points to a table; therefore, it corresponds to EdgeDescription.

In addition to establishing the correspondences, it is necessary to delimit the elements that are going to be shown through OCL queries as the ones in the List 4. From lines 1 to 4 there are queries that return a collection of forms and tables, where *self* represents a previous selected module. In addition, on line 5 there is a query that returns all the 'SingleTableRelationships' of the given application and then the technology of the graphic editor (i.e., Sirius) leaves the ones that have as source/target the forms and tables resulting from lines 1-4. Finally, in lines 5 to 11 there is a query that defines the color of a module based on the number of contained tables. For example, if the number of tables is less than or equal to a threshold, then the selected color is blue.

Listing 4 Example of OCL Queries needed in Sirius

```
1  context Module:: tables: Collection
2  self.elements.select(c | c.oclIsTypeOf(Table));
3  context Module:: forms: Collection
4  self.elements.select(c | c.oclIsTypeOf(Forms));
5  context Module:: module Color: Color
6  if self.elements ->select(oclIsTypeOf(Table)).size() <= threshold then
7  Color#Blue
8  else
9  ...
10 10 endif.
11 ;
```

Instead of developing this scaffolding, the MDRE expert has to establish the correspondences between architectural concepts and views specification by annotating the architecture metamodel. The queries from lines 1-4 are automatically generated from the annotations. The query from lines 5-11 is replaced by a snippet in the Metrics transformation (see Listing 1) and a query in the odesign file (Listing 3). Note that the query references to the measure 'Total number of Tables' previously calculated by the transformation. The snippet is manually developed by the expert and the query is automatically generated.

7.2 Limitations

The main benefit of our approach is the Independence of the technology that allows us to be flexible in terms of the language we can process and the views we can obtain. However, this benefit is, at the same time, a limitation because there is a need for an MDRE expert to build the assets which are non-trivial. However, that task has to be done only once per technology and can be reused as many applications as desired. The cost of the solution is cost-effective if final users are going to reuse the assets many times, either because it is applied in many different applications or in the same application that evolves constantly.

7.3 Related work

Our approach was designed based on two main requirements: i) software visualization; and ii) modeling of software metrics that impact the views. In this section we classify related work whose requirements are similar to ours.

7.4 Software visualization

There are several academic approaches on architectural views oriented to soft- ware comprehension tasks, for example: CodeCity (Wettel et al. 2011), eCity+ (Khan et al. 2014), AIVA (Snajberk et al. 2012; najberk et al. 2013), SAABs (Osman et al. 2014), Softwarenaut (Lungu et al. 2014), VizDSL (Morgan et al. 2018), Modigen (Gerhart and Boger 2016), EuGENia,[17] Moose,[18] and GRAPH (Bergel et al. 2014).

Several of these approaches (except by VizDSL, Modigen, EuGENia, Moose, and GRAPH) create views for a restricted number of technologies and languages; e.g., Smalltalk, Java, C++ and ADA. In the same way, the views are predefined and it is not

possible to easily define new ones. CodeCity and eCity+ generate views whose notation is based on the city metaphor: the neighborhoods represent packages and the buildings represent classes. In the case of SAABs, UML classes diagrams are obtained. In Softwarenaut, module views and their dependencies are obtained. Fi- nally, AIVA produces components and classes views. In contrast, our approximation is open from the technology point of view for several reasons. Firstly, according to the working technology, it allows new parsers to be plugged to the rest of infrastruc- ture. Secondly, it allows the definition of new views with the relevant architectural elements for the developer. Thirdly, it gives freedom to define the used notation instead of using something fixed such as UML (Snajberk et al. 2012) because not all the technologies obey the object oriented programming paradigm.

VizDSL is a visual DSL based on the Interaction Flow Modeling Language (IFML) to design and create interactive visualizations. Modigen is a textual DSL to spec- ify graphical editors DSLs based on node and edge diagrams. These are, together with an EMF metamodel, the input for a generator that produces an Eclipse-based graphical Editor. VizDSL and Modigen are generic workbenches not specially tar- geting metric-centered architectural views for software.

Like EuGENia, our tool generates an editor specification from an annotated metamodel. When using EuGENia developers define the view look-and-feel in a static manner. In contrast, when rendering our views their look-and-feel can dynamically change since Sirius takes as values of graphical properties (i.e., color, size, and labels) the measurements computed prior to visualization.

We found that our approximation resembles Moose in many aspects: it is possible to connect parsers for new technologies, there is an intermediate (meta)model on top of which queries can be made and new views can be specified. Following Moose, the same research group proposes GRAPH: A DSL to specify views that represent software dependencies. Software structural elements and their relationships are re- ferred to as nodes and edges of a graph. The views style (i.e., color, size) is defined in terms of metrics; for example, the number of methods in a class determines the size of nodes. The comparison of our work with Moose and GRAPH is explained in Section 2.

7.5 Modeling of software metrics

In (Bagnato et al. 2017), a profile of the Structure Metrics Metamodel (SMM) is contributed. The authors take advantage of Modelio tool[19] to annotate diagrams with concepts coming from the profile. Diagrams are made from scratch by final users. In contrast, our approach generates diagrams from source applications.

In (Stevanetic et al. 2014), authors define a DSL that serves to express architectural abstractions of software applications. The specification is used to track architectural changes in a given application and check compliance with a reference architecture. Authors present the language constructs to specify understandability metrics so that users can check if the current version of an application is between predefined ranges. Software visualization is out of the scope.

8 Conclusions and future work

Our approach based on annotations generates specifications of graphic editors that have architectural views that show a decomposed application from a high level to a low

level, according to the comprehension necessities based in software metrics. This allows developers to obtain graphic editors in less time, regardless of technology.

Since our experiments have shown that the editor's performance can decrease with large architecture models, a future direction is to study mechanisms that increase the performance of the queries included in the view specification model (i.e. mainly the queries that navigate the metrics model).

A DSL is a language tailored to a specific application domain. It consists of an abstract syntax, a concrete syntax, and semantics. We have targeted a more DSL- oriented approach for specifying software views. We are reaching the goal incremen- tally. We have moved a step forward by defining the abstract syntax (our TIVS and Metrics metamodels) and semantics (encapsulated in the transformations). For the time being, we are lacking a dedicated concrete syntax and the workaround is the use of annotations. The latter is because we are collecting lessons and experiences from the use of our approach with students and practitioners in order to provide a language expressive enough. However, this shortfall does not prevent the use of the approach since developers are used to employing annotations. We consider that a textual syntax would be a "nice to have" feature and, therefore, the definition of a textual concrete syntax is part of our research agenda.

Finally, another future work is to extend our approach to deal with the limitations observed in the evaluation; e.g., relationships with multiple targets.

9 Endnotes

[1]Sirius is a free platform that generates graphic editors from views specification models; we have chose it for many reasons: simplicity to specify diagrams, capacity to draw big models, and an active developers community.http://www.eclipse.org/sirius.

[2]http://composersolutions.com/

[3]http://vgosoftware.com/

[4]https://www.oracle.com/technetwork/developer-tools/jheadstart/overview/index.html

[5]https://pitss.com/

[6]http://www.renaps.com/ormit-java-adf.html

[7]http://www.sparxsystems.com/products/ea/

[8]https://www.modelio.org/

[9]https://www.ibm.com/us-en/marketplace/rational-software-architect-designer

[10]http://staruml.io/

[11]Railroady - https://github.com/preston/railroady

[12]MySQL Workbench - https://www.mysql.com/products/workbench/

[13]http://www.eclipse.org/Xtext/

[14]https://eclipse.org/MoDisco/

[15]http://www.omg.org/technology/kdm/

[16]https://wiki.eclipse.org/OCL/OCLinEcore

[17]https://www.eclipse.org/epsilon/doc/eugenia/

[18]http://moosetechnology.org/

[19]https://www.modelio.org/

Abbreviations

CRUD: Create, read, update and delete; EJB: Enterprise java bean; JEE: Java enterprise edition; KDM: Knowledge discovery metamodel; LOC: Lines of code; MDRE: Model-driven reverse engineering; MVC: Model-view-controller;

OMG: Object management group; PIM: Platform independent metamodel; PL/SQL: Procedural language/structured query language; RoR: Ruby On rails; SMM: Structured metrics modetamodel; UML: Unifired modeling language

Acknowledgements
The authors thank Diego Castiblanco for helping in the translation of the article.

Funding
The research was funded by Universidad de Los Andes.

Availability of data and materials
Please contact authors for data requests.

Authors' contributions
LFM contributed in the design of the approach. He fully developed the tool and the experiments. KGP stated the problem out and participated in the design of the approach. RC helped to draft the manuscript. All authors contributed writing, reading and approving the final manuscript.

Authors' information
LFM is software architect of the Technology, Networking and Information Systems Management Office at Universidad de Ibagué. Prior to this, he was assistant researcher in the Department of Systems and Computing Engineering at Universidad de los Andes, where he finished his master degree in Software Engineering.
KGP is assistant professor of the Department of Systems and Computing Engineering at Universidad de los Andes. Prior to this, she was R & D Engineer / Software Engineer at Netfective Technology SA. She received her Ph.D. in September 2010 from the University of Nantes. In 2011, she was a post-doctoral fellow in INRIA lab. She has participated in research and development projects (proprietary or Open Source) since 2005. Her research interests are software engineering, evolution and maintenance of software and model-driven engineering.
RC is full professor of the Department of Systems and Computing Engineering at Universidad de los Andes. Her research interest are: Software development based on models, Software product lines, and Modeling specific domains.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Department of Systems and Computing Engineering, School of Engineering, Universidad de los Andes, Cra 1 No 18A - 12, Bogotá, Colombia. [2]Gestión de Tecnologías, Redes y Sistemas – GTRES, Universidad de Ibagué, Cra 22 Cl 67 B/ Ambalá, Ibagué, Colombia.

Received: 25 March 2018 Accepted: 20 November 2018
Published online: 04 December 2018

References

Anquetil N, Laval J (2011) Legacy software restructuring: Analyzing a concrete case. 15th European Conference on Software Maintenance and Reengineering. IEEE, Germany, p 279–286
Bagnato A, Sadovykh A, Dahab S, Maag S, Cavalli A, Stefanescu A, Rocheteau J, Mallouli S, Mallouli W (2017) Modeling omg smm metrics using the modelio modeling tool in the measure project
Bergel A, Maass S, Ducasse S, Girba T (2014) A domain-specific language for visualizing software dependencies as a graph. 2014 Second IEEE Working Conference on Software Visualization. IEEE, Canada, p 45–49. https://doi.org/10.1109/VISSOFT.2014.17
Brunelière H, Cabot J, Dupé G, Madiot F (2014) Modisco: A model driven reverse engineering framework. Inf Softw Technol 56(8):1012–1032. https://doi.org/10.1016/j.infsof.2014.04.007
Czarnecki K, Eisenecker UW (2000) Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York
Escobar D, Cárdenas D, Amarillo R, Castro E, Garcés K, Parra C, Casallas R (2016) Towards the understanding and evolution of monolithic applications as microservices. 2016 XLII Latin American Computing Conference (CLEI). IEEE, Chile, p 1–11. https://doi.org/10.1109/CLEI.2016.7833410
Garcés K, Casallas R, Álvarez C, Sandoval E, Salamanca A, Viera F, Melo F, Soto JM (2018) White-box modernization of legacy applications: The oracle forms case study. Comput Stand Interfaces 57:110–122
Garcés K, Sandoval E, Casallas R, Alvarez C, Salamanca A, Pinto S, Melo F (2015) Aiming Towards Modernization: Visualization to Assist Structural Understanding of Oracle Forms Applications. ICSEA 2015, Tenth International Conference on Software Engineering Advances. IARIA, Spain, p 86–95
García J, Garcés K (2017) Improving understanding of dynamically typed software developed by agile practitioners. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.ESEC/FSE 2017. ACM, Germany, p 908–913. https://doi.org/10.1145/3106237.3117772
Gerhart M, Boger M (2016) MODIGEN: Model-Driven Generation of Graphical Editors in Eclipse. Int J Comput Sci Inf Technol 8;73-91. https://doi.org/10.5121/ijcsit.2016.8506

Khan T, Humayoun SR, Amrhein K, Barthel H, Ebert A, Liggesmeyer P (2014) eCITY+: A Tool to Analyze Software Architectural Relations Through Interactive Visual Support. Proceedings of the 2014 European Conference on Software Architecture Workshops. ECSAW '14, New York, pp 36–1364. https://doi.org/10.1145/2642803.2642839
Lungu M, Lanza M, Nierstrasz O (2014) Evolutionary and Collaborative Software Architecture Recovery with Softwarenaut. Sci Comput Program 79:204–223. https://doi.org/10.1016/j.scico.2012.04.007
Mancoridis S, Mitchell BS, Chen Y, Gansner ER (1999) Bunch: a clustering tool for the recovery and maintenance of software system structures. Proceedings IEEE International Conference on Software Maintenance, 1999. (ICSM '99). IEEE, England, p 50–59
Mendivelso, L. Garcés, K. Casallas, R. Vistas arquitectónicas independientes de tecnología para compresión de software. Proceedings of the 20th Conferencia Iberoamericana en Software Engineering (CIbSE 2017). Buenos Aires, pp 71-84
Minelli R, Mocci A, Lanza M, Kobayashi T (2014) Quantifying program comprehension with interaction data. 2014 14th International Conference on Quality Software. IEEE, USA, p 276–285. https://doi.org/10.1109/QSIC.2014.11
Morgan R, Grossmann G, Schrefl M, Stumptner M, Payne T (2018) VizDSL: a visual DSL for interactive information visualization. In: Krogstie J, Reijers H (eds) Advanced Information Systems Engineering. CAiSE 2018. Lecture Notes in Computer Science, vol 10816. Springer, Cham
Najberk J, Holy L, Brada P (2013) Visualization of component-based applications structure using AIVA. 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, Italy, p 409–412. https://doi.org/10.1109/CSMR.2013.60
OMG: Structured Metrics Metamodel (SMM). http://www.omg.org/spec/SMM/1.0/ Accessed 08 Mar 2018
Osman MH, Chaudron MRV, van der Putten P (2014) Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams, vol 1. 2014 21st Asia-Pacific Software Engineering Conference. IEEE, South Korea, p 159–166. https://doi.org/10.1109/APSEC.2014.34
Rugaber S, Stirewalt K (2004) Model-driven reverse engineering. IEEE Softw 21(4):45–53. https://doi.org/10.1109/MS.2004.23
Snajberk J, Holy L, Brada P (2012) AIVA vs UML: Comparison of component application visualizations in a case-study. 2012 16th International Conference on Information Visualisation. IEEE, France, p 54–61. https://doi.org/10.1109/IV.2012.20
Stevanetic S, Haitzer T, Zdun U (2014) Supporting software evolution by integrating DSL-based architectural abstraction and understandability related metrics. ACM International Conference Proceeding Series. ACM, Austria. https://doi.org/10.1145/2642803.2642822
Tilley S (2009) Documenting software systems with views VI: lessons learned from 15 Years of research & practice. Proceedings of the 27th ACM International Conference on Design of Communication. SIGDOC '09, New York, pp 239–244. https://doi.org/10.1145/1621995.1622043
Wettel R, Lanza M, Robbes R, Software Systems A (2011) Cities: a controlled experiment. Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, New York, pp 551–560. https://doi.org/10.1145/1985793.1985868
Wikipedia: Creative Commons Attribution-ShareAlike License. Page Version ID: 755202644 (2016)
Wikipedia: Saffir–Simpson scale. Page Version ID: 786709578 (2017). https://en.wikipedia.org/w/index.php?title=Saffir%E2%80%93Simpson_scale&oldid=786709578 Accessed 29 June 2017