Aiming towards Modernization: Visualization to Assist Structural Understanding of Oracle Forms Applications

Kelly Garcés, Edgar Sandoval,
Rubby Casallas, Camilo Álvarez
Los Andes University
School of Engineering
Department of Systems and Computing Engineering
Bogotá, Colombia
email:{kj.garces971,ed.sandoval1644,rcasalla,c.alvarez956}
@uniandes.edu.co

Alejandro Salamanca, Sandra Pinto, Fabian Melo
Asesoftware
Bogotá, Colombia
email:{asalaman, spinto, fmelo}@asesoftware.com

***Abstract*—Oracle Forms is a tool for creating screens that interact with an Oracle database. It appeared in the eighties and its use spread to many IT sectors today. There are pressures that push software engineers to modernize Oracle Forms applications: obsolescence of technology, requirements of users, etc. For a straightforward modernization, it is necessary to comprehend the applications from a prior step. This paper reports the preliminary results of the "Forms Modernization" project, in particular, of the understanding step. In most cases, the understanding of Forms applications is a complex and time-consuming task due to several reasons: large size of applications, lack of design documentation, lack of software organization. This paper proposes a visualization process to alleviate these issues. The process takes static Oracle Forms code as input and produces a set of domain specific diagrams/views, that ranges from high to low abstraction levels, as output. The gist of diagrams and views is to assist engineers in a structural understanding of the Oracle Forms software. The process includes algorithms for element discovery and clustering, and is instrumented by means of a tool running on Eclipse Modeling technologies. We take advantage of four real Oracle Forms applications to illustrate the benefits of this approach. These applications have been provided by Asesoftware, which is the Colombian industrial partner of the project.**

Keywords*—*program comprehension; reverse engineering; tools; clustering algorithms; model-driven engineering; graphical editors.

I. Introduction

Software is constantly evolving; this evolution is motivated by different reasons such as the obsolescence of a technology, the pressure of users, or the need to build a single coherent information system when companies merge [1]. Our research lies in the field of software modernization, a kind of evolution, that refers to the understanding and evolving of existing software assets to maintain a large part of their business value [2].

This paper presents the preliminary results of the "Forms Modernization" project, which involves academic and industrial partners. The project arose as a result of some problems faced by Asesoftware, a Colombian software company that offers modernization services to its clients, regarding the desire to migrate Oracle Forms applications to modern platforms (in particular Java).

Oracle Forms appeared towards the end of the 1980s. It comprises a rapid database application development environment and a runtime environment, where the database applications run. Oracle Forms applications are present in many sectors. Such is the case in Colombia as well as in other countries. Results of a tool usage survey [3], carried out by the Oracle User Group Community Focused On Education (ODTUG) in 2009, indicate that 40 percent of 581 respondents (application developers) use Oracle Forms.

The migration of Oracle Forms applications to new technologies is mainly caused by three factors: the fear that Oracle desupports Forms, the difficulty to find Forms programmers, and Forms no longer meeting business requirements.

Furthermore, the company, Asesoftware, complains about the following three problems of manual modernization: i) Difficulty to understand the Oracle Forms application, ii) Time-consuming and repetitive migration, and iii) Poor testing. The "Forms Modernization" project addresses these problems in three phases. Here, we report the results of the phase that aims to solve the first problem.

According to Lethbridge and Anquetil [4], the main difficulties when trying to understand legacy applications are the following: i) lack of a directory hierarchy and of design information, ii) original designers' lack of knowledge of software architecture, and iii) undermining of the original design decisions as many additions and alterations were made. An Oracle Forms system is not the exception to Lethbridge and Anquetil's claim about legacy software organization: it lacks a directory hierarchy and the file names are not necessarily meaningful. As a result, an inspection of this code, aimed at understanding, is time consuming and error-prone.

To cope with this, Asesoftware organizes meetings with the clients, where the latter transfer their knowledge regarding application functionalities to the engineers in charge of the modernization process. The purpose of these meetings is to obtain a global understanding of the application's functional requirements, in order to ease the subsequent inspection of the code as well as the migration process. Nevertheless, this understanding remains in the mind of the engineers and it is not reported in any formal document in a way that the learning curve could be shortened for new people that enter the modernization process.

This paper presents the proposed approach as a solution to

Copyright (c) IARIA, 2015.

the understanding issue. The approach consists of a process that takes a given Oracle Forms application as input and produces a set of diagrams and views that give an insight into the application's structural organization as output. The process includes algorithms for element discovery and clustering, and is instrumented by means of a tool running on Model-Driven engineering technologies. The resulting diagrams and views are designed to satisfy three concrete understanding challenges. When comparing our approach to related work —either research work [5][6][7] or commercial tools (i.e., Oracle2Java [8], Evo [9], Jheadstart [10], Pitss [11], Ormit [12])— we found that they only provide views with a low level of abstraction, whereas our approach proposes diagrams and views that range from high to low abstraction levels, thus, contributing to the acceleration of the understanding of the Oracle Forms program and aiming at modernization.

The paper is structured as follows: Section II describes the main building blocks of Oracle Forms applications and introduces four real Oracle Forms applications that serve as illustrating examples. Through an example, Section III elaborates on the understanding challenges that guide our research. Section IV establishes certain criteria, classifies related work according to it, and compares these works to our proposal. Sections V and VI present our approach and the tool used for instrumentation, respectively. Section VII describes how the user interacts with the visualizations in order to achieve an understanding. Section VIII elaborates on the results of applying our proposal to the illustrating examples. Finally, Section IX concludes the paper and outlines future work.

II. Oracle Forms Overview and Illustrating Examples

We present the main concepts of an Oracle Forms application below:

- Form: A Form is a collection of objects and code, including windows, items, triggers, etc.
- Blocks: Represent logical containers for grouping related items into a function unit to store, display and manipulate records of database tables. Programmers configure the blocks depending on the number of tables from which they want to manipulate the form:
  - The way to display a single database table in a form is to create a block. This results in a single table relationship between the form and the table.
  - The way to display two tables that share a master-detail relationship (i.e., "One to Many" relationship) is through two blocks. Oracle Forms guarantees that the detail block will display only records that are associated with the current record in the master block. This results in a master/detail relationship between the form and the two tables.
- Item: Items display information to users and enable them to interact with the application. Item objects include the following types: button, check box, display item, image, list item, radio group, text item and/or user area, among others.
- Trigger: A trigger object is associated to an event. It represents a named PL/SQL function or procedure that is written in a form, block or item. PL/SQL is the Oracle procedural extension of SQL. PL/SQL allows programmers to declare constants, variables, control program flows, SQL statements and APIs. A useful Oracle Forms API written in PL/SQL is the one offering procedures for form displaying, i.e., the OPEN/CALL statements.
- Menu: Is displayed as a collection of menu names appearing horizontally under the application window title. There is a drop-down list of commands under each menu name. Each command can represent a submenu or an action.

These concepts are found in the examples that will be used throughout the paper. These examples are aligned with four real applications related to treasury, banking and insurance sectors. These applications will be referred to as Conciso, Servibanca, Maestro, and Sitri. The following information is useful in order to give an idea about the application's size: 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.

III. Challenges Illustrated by an Example

Using a concrete example, this section presents the challenges we face. Suppose a form of Conciso has to be modernized in two senses: i) evolution towards a new technology, and ii) introduction of a small modification to the initial functionality. The form allows manipulating deductions from an Employee's withholding tax. The modification consists in taking into account the deductions to which an employee has the right after making donations to institutions that promote culture, sports and art at a municipal level. Specifically, this modification should ensure that the user indicates a city, department and country in the form when the option of deduction by donations to local institutions has been chosen.

We face the following challenges as we try to understand the scope of the modernization at an application level:

A. Challenge 1: Functional modules and their relationships

This challenge concerns the following questions: *What is the functional module that contains the form subject to modernization? Is this module related to another modules?*

The fact of knowing the module that contains the form subject matter of modernization helps engineers to delimit the modernization scope. As we said in the introduction Section,the Oracle Forms software often lacks documentation, directory hierarchy and meaningful naming conventions; as a consequence, the functional modules are hard to infer. This is the case in the scenario where the client provides a folder that contains 144 forms on the root, with no subfolders nor documentation. Each form has a name that is the concatenation of a prefix (e.g., *CBF*) and a 5-digit number. In addition, the Oracle forms IDE only shows one form at a time, so that there is no a notion of a forms container.

Furthermore, it is important to know the dependency relationships between modules. A dependency relationship

between modules results when forms $a_1, a_2, .., a_n$ call forms $b_1, b_2, .., b_n$, and the forms are contained in two different modules A and B. Engineers can use this kind of relationship as an indication about the potential impact that a modification in the deduction form has on forms that belong to different modules. As for the modules, it is also hard to derive the relationships between modules in Oracle Forms. To do so, it is necessary to inspect the form PL/SQL code and look for CALL/OPEN statements directed towards another form. Note that these statements are spread along the form elements, which makes it difficult to discover the relationships between modules.

B. Challenge 2: Relationships between forms and tables

When addressing this challenge one should be able to answer the following questions: *Which are the tables related to the form that will be modernized? What is the type of relationship between the form and the tables?*

The relationships between forms and tables are important because they suggest to engineers that they have to review, more in detail, how changes in tables (for example, adding a foreign key from the deduction table towards the city table) impact form elements and their embedded PL/SQL code. The amount of effort to find out the tables related to a form depends on the type of relationship. Whereas single table and master/detail relationships are relatively easy to discover, by regarding the form navigation tree available on the Oracle Forms IDE, relationships resulting from references to tables embedded into the PL/SQL code are more time demanding because the code is scattered throughout the form elements (i.e. forms, blocks, items).

C. Challenge 3: Relationships between forms

This challenge includes providing an answer to the question: *Is there any form related to the form that will be modernized?* The purpose of this question is twofold: i) to know if related forms have to be changed in order to fully satisfy the functionality of the form after its modernization, and ii) to figure out if changes to the form subject matter of modernization impact the capabilities of the related forms. The example mentioned at the beginning of this section illustrates the first part of the purpose: it is important to know if there is any form —currently calling the deduction form— that needs to be modified in order to specify the different options of deductions and display the deduction form in an appropriate manner by taking into account the selection made by the user. This challenge is related to the first one in the sense that the relationship between two modules depends on the relationships between the forms that are contained in the modules. The mechanism to infer the relationships between forms is to review the PL/SQL code seeking for CALL/OPEN statements. Because this task has to be performed regardless of whether the forms are in the same module or in two different modules, it is very time consuming.

The challenges above are valid for multiple scenarios; they motivate the approach we propose in Section V. However, before elaborating on the approach, we present related work that helps us establish a background regarding visualization processes.

IV. Related Work

In this Section, we establish criteria that help us classify related research. For each criterion, we give a definition, variations on how the criterion can be satisfied —which results in categories—, and the position of each related work within these categories. The Section ends by comparing our approach to those found in related work.

A. Software systems

Tilley [13] has conducted extensive research into the use of views as a form of program comprehension. He found that numerous approaches focus on three different categories of software: i) legacy systems, ii) web applications, and iii) application design patterns. After considering the results of referential databases, we resolved that Tilley's criterion to classify view-related works is still valid, and decided to use it in our classification.

The legacy systems category encompasses traditional systems characterized as follows: monolithic, homogeneous, and typically written in third generation procedural programming languages. The purpose of related work within this category [5][6][7] is to achieve an understanding of the system, that can serve as a basis for its maintenance or for migration to newer languages.

The second category comprises Web applications. These systems often share many of the negative features of traditional legacy systems (e.g., poor structure, little or no documentation). The gist of related work in this category [14][15] is to understand the interaction behavior of the Web application, for further development and maintenance.

Finally, the third category covers a broader range of systems, including the software systems mentioned above. However, the difference is that related research within this category [16] specializes on design pattern recognition for better comprehension. The provided views are important to detect the critical points of an application for maintenance purposes.

B. Process

This criterion describes the steps that are followed to generate software systems views. Most of the reviewed approaches [5][6][7][14][16] agree with the following three steps: i) data injection, ii) querying, and iii) visualization. The first step consists in obtaining an in-memory representation from the input software artifacts. The second step aims at building blocks through the representation. Finally, the third step includes the generation of views for the groupings of blocks that result from the second step.

C. Input

This criterion indicates which kinds of inputs can be used by the process mentioned above. Literature reports mainly two kinds of inputs: i) Static input, and ii) Dynamic input. A static input only refers to source code [5][6][7]. Dynamic inputs, in contrast, are related to run-time information. Authors [14][15][16] obtain the second kind of input by executing scenarios that help them identify the invocation of a specific software feature (e.g., field, method, web page). Commonly, dynamic inputs are complemented by static inputs.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

D. Source code language

This criterion points out the languages in which the source code is written; there are two categories: i) language specific, and ii) language independent. While the first category classifies the works whose implementations can be applied to source code written in a particular programming language, the second category encompasses the tools that can be applied to a variety of languages. Most of works [5][6][7][14][16] fall in the first category, they reverse engineer applications written in PHP, COBOL, Smalltak, C++, Oracle Forms. In contrast, there are few approaches in the second category [17]. The strategy of the latter is to develop bridges (i.e., programs, compilers, grammars, transformations) that allow the authors to go from the source code to an intermediate format on top of which the views are built up.

E. Notation

This criterion determines whether the notation used in the view is: i) standard, or ii) domain specific. The second option is suggested over the first one in cases where the reverse engineering task includes experts/users for which a customized graphical notation results in straightforward comprehension and communication. However, the second option implies a higher development effort when compared with the first one: while the first option can reuse existing viewers, the second option often requires the construction of viewers from scratch. The most disseminated standard notation among the articles [5][14][17][18] is the Unified Modeling Language (UML), in particular, class and sequence diagrams. Another popular standard notation is the graph theory, where nodes and edges are generic enough to represent any kind of software element and relationship between elements. An example of an articles that uses graph notation is [16]. In turn, the following are articles that propose domain specific notations: [6][7].

F. Views

In this criterion, we take advantage of Lowe's taxonomy [19] to classify the proposed views according to related work. Lowe et al. arrange the views in two categories: i) high level, and ii) low level. The high level category covers the views suitable to directly support program comprehension. Examples of such representations are class interaction graphs, lists of possible components/modules/subsystems, and architectural diagrams. On the other hand, low level views are much too complex to provide any understanding of any non-trivial program. Examples of low level representations are basic block graphs, single static assignment representations, call graphs, and control flow graphs. The following are some related works that provide high level views: [5][6][14][16][18]. In turn, [7][18] fall on the low level view category.

G. Comparison

Taking into account the criteria mentioned above, we compared our approach to related work —research work and commercial tools included— and we reached the following conclusions:

- *Software systems*: Similarly to [5][6][7], our approach falls on the legacy system category.

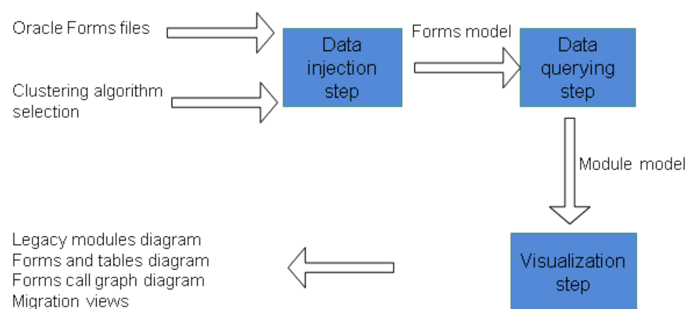


Figure 1. View-generation process overview

- *Process*: Our solution overlaps all previously cited solutions in the three steps of the view-generation process.
- *Input*: In similar fashion to [5][6][7], our approximation takes only source code as input.
- *Source code language*: Our approach takes source code written in Oracle Forms as input. That is, it belongs to the language specific category as well as [5][6][7].
- *Notation*: Like [6][7], our solution proposes a domain specific notation.
- *Views*: There is a noticeable difference between our approach and related work with respect to this criterion. Our literature review points out [7] as the sole scientific approach that provides views for Oracle Forms program understanding. The review also includes a set of commercial tools (i.e., Oracle2java, Evo, Jheadstart, Pitss, Ormit) that propose views for the same purpose. In both cases, the views are of two kinds: i) layout view and ii) application navigation tree. The layout view reflects how the graphical elements are arranged on a form and displayed to the user. The application navigation tree provides a hierarchical display of all forms in an application as well as the objects in each form —triggers, blocks, program units, etc.—. In our opinion, these views would be categorized as low level since they show a high level of detail; in contrast, we provide not only low level views but also high level ones, which can accelerate the understanding of the Oracle Forms program.

V. VIEW-GENERATION PROCESS

The view-generation process (see Figure 1) involves reverse engineering the Oracle Forms application and presenting several different diagrams and views to the developers. These diagrams and views can be further analyzed to determine subsystems, the elements that compose these subsystems(e.g., forms, database tables), and the relationships between these elements.

A. Data injection step

This step corresponds to data injection, which is the first step of a classical view-generation process (see Section IV-B).

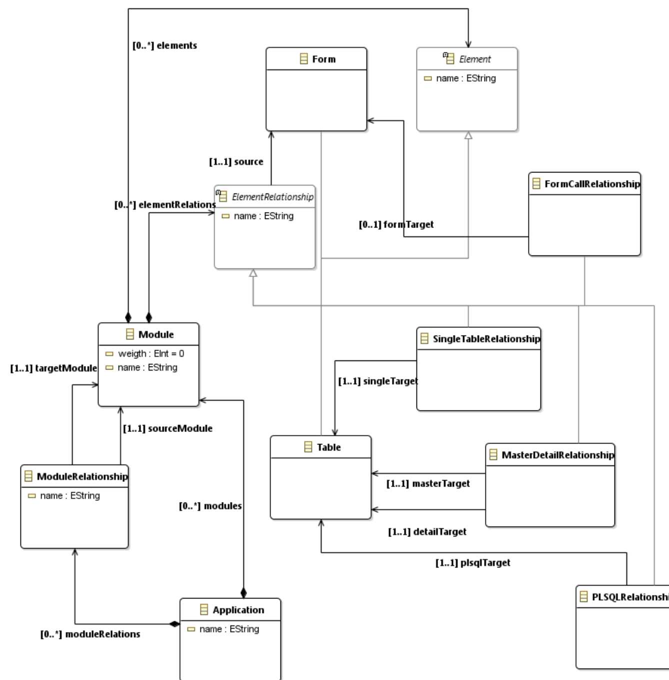


Figure 2. Module metamodel

As mentioned before, the purpose of this step is to obtain an in-memory representation from the input software artifacts. In our approach, we obtain an Abstract Semantic Graph (ASG) from Forms files (.fmb, .mmb). While a .fmb file describes a particular form, a .mmb file describes the menu from which all the application forms are displayed. This ASG is navigated to create model elements that conform to the Form metamodel. The main concepts of this metamodel have already been mentioned in Section II, namely forms, menus, blocks, items, triggers, relationships and tables. It is worth noting that we manage to extract not only the tables that are directly referenced by blocks, but also table references embedded into PL/SQL code. The reason to use models instead of the ASG is that we use tools that easily build editors for diagrams and views on top of models (see SectionVI).

B. Data querying step

This step corresponds to the second step of the view-generation process, whose gist is to search for building blocks through the representation. In our case, the representation is the Form model mentioned in the previous step and we search for elements such as modules, forms, tables, and relationships. Then, the elements resulting from this search are represented in another model, referred to as *Module model*. In contrast to the former model —which is verbose—, the latter model contains only the elements that matter in the visualization step. We describe the main concepts of the module model below and, then, the algorithms used to obtain it.

1) Module model: This model conforms to a metamodel (see Figure 2) and its concepts are explained below:

- *Application* is the root element of the metamodel. It describes the Oracle Forms application under study. An application consists of a set of modules that are related to each other.
- *Module* is a necessary concept because it works as a container of Oracle Forms elements and their relationships.
- *ModuleRelationship* represents the relationship between a pair of modules. A relationship going from module A to module B means that A contains a form that calls a form from B.
- *Element* describes Oracle Forms elements, i.e., forms and tables.
- *Form* specifies an Oracle form.
- *Table* indicates a table referenced from a form.
- *ElementRelationship* represents a relationship between a pair of Oracle Forms elements. An ElementRelationship can be classified into one of the following four sub-kinds.
- *SingleTableRelationship* is established between a form and a table, when the form has a block that references that table.
- *MasterDetailRelationship* is established between a form and two tables, when the form has two blocks —related via properties—, the tables are those referenced by the blocks, and one of them has the master role and the other one the detail role.
- *PLSQLRelationship* is established between a form and a table, if the form has PL/SQL that contains occurrences of the table name. A classic example of this is the relationship created from a form with a lookup field. The form contains a block with the addition of one field that displays data from another table. Such data is "looked up" via PL/SQL code when the form runs.
- *FormCallRelationship* is established between two forms C and D, if form C contains CALL/OPEN statements parametrized with the name of form D.

2) Algorithms: Two kinds of algorithms are necessary to obtain a given module model:

1) *Element discovery algorithm*: This algorithm creates appropriate model elements depending on the forms, tables, and relationships found in the Forms model.
2) *Clustering algorithms*: One of the main concepts in the metamodel is the *Module*. Having modules makes the software easy to understand and, therefore, to change. However, it is not always easy to get the modules because legacy software organization is often quite poor. To cope with this, previous works in software comprehension[20][21] have used clustering algorithms. A clustering algorithm arranges software components into modules, by evaluating the relationships among these components. We have implemented the following two clustering algorithms that arrange the forms, tables and relationships discovered by the Element discovery algorithm, into modules:

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

a) Menu-based clustering algorithm: This algorithm takes a Forms model and its corresponding Module model —which results from the Element discovery algorithm— as inputs. From these two inputs, the menu-based clustering algorithm is in charge of producing a new Module model where the model elements (i.e., forms, tables, and relationships) are arranged into modules. For each menu in the Forms model, the algorithm inspects the commands in the respective drop-down list until it reaches the commands that are calls to forms. Then, the algorithm creates a module element —whose name is the corresponding menu name— and groups each form element within the module, according to the form name indicated by the corresponding call. In addition, the algorithm arranges the tables into the modules, following the relationships existing between forms and tables. Asesoftware Oracle Forms experts argue that there is good accuracy in the resulting modules diagrams when looking at the menus; however, they also point out that there is a lack of .mmb files because Oracle Forms programmers prefer to create menus by manually adding buttons through a .fmb file. We propose the following algorithm to tackle this lack of .mmb files.

b) Table betweenness clustering algorithm: This algorithm has four phases:

1) In the first phase, it takes a Module model —which results from the Element discovery algorithm— as input and produces a graph as output. In the graph, the nodes represent forms, and an edge is established between each pair of nodes (or forms) if they have several tables in common.
2) In the second phase, the algorithm determines the modules, that is, the subgraphs of the graph obtained in the first phase. This algorithm identifies a subgraph because its inner connections are dense, and the outer connections among subgraphs are sparser. There are several ways of identifying subgraphs, however, due to the delivery dates of the project being so close, we decided to use an existing method: the Girvan-Newman algorithm [22]. Therefore, in the second phase, our algorithm delegates the subgraph construction to the Girvan-Newman algorithm. The latter progressively finds and removes edges with the highest *betweenness*, until the graph breaks up into subgraphs. The betweenness of an edge is defined as the number of shortest paths between all pairs of nodes in the graph passing through that edge. If there is more than one shortest path between a pair of nodes, then each path is assigned equal weight such that the total weight of all of the paths is equal to unity; nonetheless, the betweenness value for an edge is not necessarily an integer. Because the edges that lie between subgraphs are expected to be those with the highest betweenness values, a good separation of the graph into subgraphs can be achieved by removing them recursively.
3) In the third phase, our algorithm creates a module element for each subgraph indicated in the Girvan-Newman algorithm output. For each node in a subgraph, the algorithm groups into the module the corresponding form element. To do so, the algorithm follows two rules: i) If a subgraph has more than one node (i.e., a form), the algorithm arranges the forms (and referenced tables) within a new module —whose name is the concatenation of a keyword and a counter—; ii) If a subgraph has only one node, then it is arranged into the *isolated form module*.
4) Finally, in the fourth phase, the algorithm arranges the tables into the modules, by following the existing relationships between forms and tables.

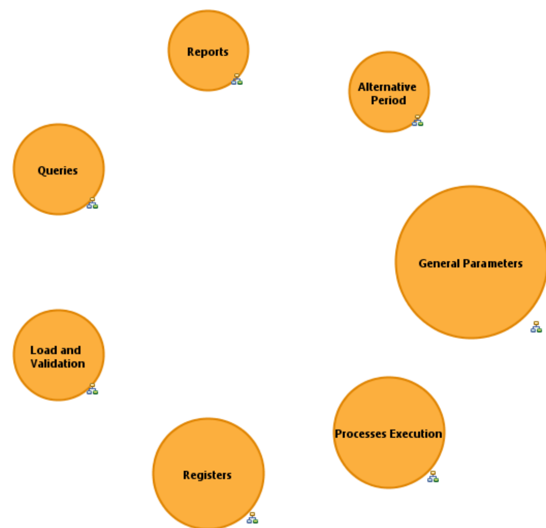


Figure 3. Legacy modules diagram for Conciso (result of the Menu-Based clustering algorithm)

It is worth noting that the number of database tables in common and the number of iterations of the Girvan-Newman algorithm, that is, the parameters used in the first and second phases, respectively, are given by the user and impact the number of resulting modules as follows: A highest number of database tables in common or a highest number of iterations result in the following: i) a highest number of modules, which are small in size because each of them contains few forms, and ii) an big-sized isolated form module that contains a lot of forms.

C. Visualization step

This step involves techniques that are of use to present the gathered information via diagrams and views. These diagrams and views are high-level or low-level representations that allow developers to obtain a structural understanding of the system. Basically, the diagrams have nodes and edges, and the views look like tables. We describe the different aspects of diagrams below: category (either low or high), purpose, notation, layout and filters that ease their navigation. The Section ends by presenting the information displayed in the table-like views.

1) Functional modules and their relationships: This diagram belongs to the high-level category. Its main purpose is to show how a legacy system is organized in terms of modules or subsystems, and which are the relationships between the modules of a system. A secondary purpose of the legacy module diagram is to serve as an entry point for the *Forms and tables diagram*. Figure 3 shows the diagram for Conciso, after applying the menu-based clustering algorithm (the notation is

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

the same as in the table betweenness algorithm). The notation used in both diagrams is explained below:

- An orange circle represents a Module. The circle label is the module name, which can be changed by the user into a more meaningful name. The size of the circle is proportional to the number of form elements contained within the module.
- A red arrow represents a ModuleRelationship.

The modules are radially arranged in descending order by size. The module with the biggest size is positioned at three-o'clock and the remaining modules are organized proceeding clockwise. In addition, the diagram provides a filter that hides ModuleRelationships.

2) Forms and tables diagram: This diagram falls into the low-level category. It is available when one selects a module from the legacy modules diagram. Its purpose is to show the forms and tables contained in the module and the relationships between them. Figure 4 shows excerpts from the Forms and tables diagram of a module of the illustrating example (i.e., *General Parameters*). The notation that was used is explained below:

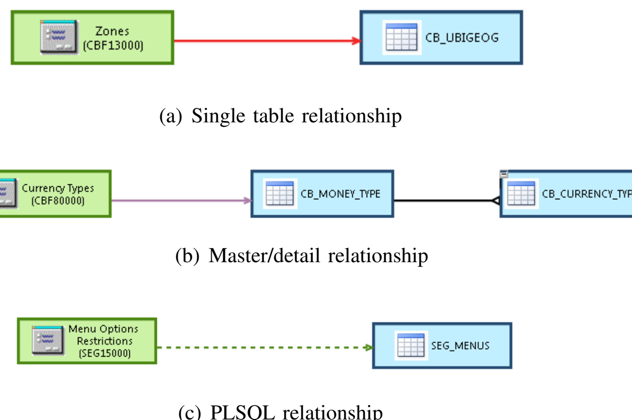
- A green square represents a Form. The square label is the form name (if present) or the file name that corresponds to the form.
- A blue square depicts a Table. The label is the table name.
- A red arrow indicates a SingleTableRelationship (see Figure 4(a)).
- A pair of purple and black arrows indicates a MasterDetailRelationship. In particular, the purple arrow points to the master table and the black arrow to the detail table (see Figure 4(b)).
- A green dotted arrow represents a PLSQLRelationship (see Figure 4(c)).

The diagram layout is in charge of placing all the elements in a way that the relationships intercept as little as possible. Furthermore, the diagram offers filters that allow us to leave all the relationships of a certain type visible in the diagram.

3) Forms call dependency diagram: This diagram belongs to the low-level category. This diagram presents the call-graph of the forms of an Oracle Forms application. Figure 5 shows an excerpt from the Forms call dependency diagram of the *General Parameters* module. The notation that was used is explained below:

- A green circle represents a Form. The circle label is a concatenation of the form name (if present) and the file name that corresponds to the form.
- The arrows describe FormCallRelationships between forms. In particular, a green arrow indicates an OPEN statement and a red arrow a CALL statement.

The forms are arranged following a tree layout. In addition, there are two kinds of filters: i) A filter that removes all unconnected Forms from the diagram, and ii) A filter that, if disabled, hides all FormCallRelationships.



(a) Single table relationship

(b) Master/detail relationship

(c) PLSQL relationship

Figure 4. Excerpt of Forms and tables diagram for Conciso.

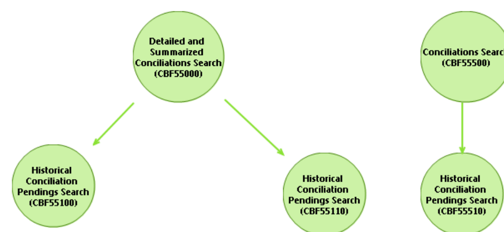
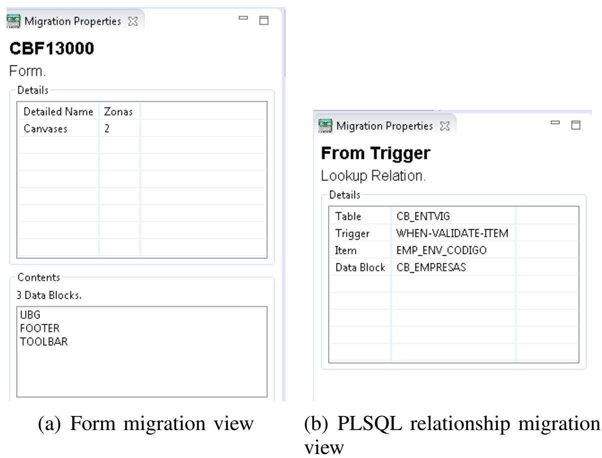


Figure 5. Forms call dependency diagram for Conciso.

4) Migration views: This view falls into the low-level category. It displays detailed information about an element, when it is selected by the user from one of the aforementioned diagrams.

- The *Module migration view* is displayed when a module is selected from the legacy module diagram. It shows the module's weight and the forms and tables it contains. Due to page restrictions, a figure illustrating this view is not included. It is worth noting that this view looks like the views below, but it displays different information.
- The *Form migration view* is shown when a form is chosen from the forms and tables diagram. It demonstrates the detailed form name, the number of canvases, and the blocks and program units declared in the form (see Figure 6(a)).
- The *Relationship migration view* is offered when a relationship is selected from the forms and tables diagram. The view shows the relationship details according to its type:
  - In case of a MasterDetailRelationship, it points out the master and detail tables, the Oracle Form relationship, and the block.
  - In case of a SingleTableRelationship, it shows the table and the corresponding block.
  - In case of a PLSQLRelationship, it shows the table, the respective block, and the trigger where the PL/SQL is embedded (see Figure

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1



(a) Form migration view (b) PLSQL relationship migration view

Figure 6. Migration view examples for Conciso.

6(b) that contains the most complex migration view for relationships).

VI. TOOLING

We have built a tool that instruments our approach. The components that comprise the tool architecture are described below. Components 1 to 4 are part of an existing open source infrastructure[23] that we used to build the tool, but components 5 to 10 are built by us.

1) *Eclipse*: includes a basic platform (i.e. workbench, workspace and team facilities) that is useful for the development of extensions.
2) *EMF*: is a framework for building tools based on a metamodel.
3) *Acceleo*: includes a feature that interprets the Object Constraint Language (OCL). OCL is a language that provides query expressions on any model.
4) *Sirius*: is a plug-in to create graphical editors that allow edition and visualization of models. Sirius can be classified into the approaches provided with DSL constructs that serve to specify the graphical notation of a view, e.g., rules to determine color/size of nodes or edges, layout, etc. Sirius is applicable to any domain, for example, a Sirius node can represent a software system entity but also the member of a family.
5) *Domain metamodels*: contains the Forms and Module metamodels presented in Section V.
6) *Forms injector*: its purpose is twofold: i) to obtain a form model from Forms files, and ii) to enrich the Module model with PL/SQL relationships. In order to meet the first purpose, we take advantage of the JDAPI [24], which is an API to manipulate Forms files. Thus, we navigate the ASG —resulting from the JDAPI— and create model elements according to the Forms metamodel classes. To attain the second purpose, we adapt an existing PL/SQL ANTLR parser.
7) *Clustering algorithms*: these algorithms have been implemented as Java programs. In particular, the table betweenness clustering algorithm takes advantages of the betweenness centrality algorithm that comes with the JUNG API [25].
8) *Diagrams and views specification*: A model that, conforming to Sirius constructs, specifies the graphical notation of diagrams and views.
9) *Customized layout*: implements layouts beyond Sirius' default layouts (i.e., tree or composite). Currently, it contains the radial layout implementation, useful in the legacy modules diagram.
10) *Wizard*: is a graphical interface that allows engineers to configure visualization process aspects, that is, the Form files path, the form to be processed and the clustering algorithm.

VII. INTERACTION PATH

In this Section, we describe how the engineer, guided by a set of questions —stated in the form of challenges— uses the visualizations in order to achieve a progressive understanding of the Oracle Application. Like in previous sections, Conciso is used for illustration purposes.

A. Challenge 1: Functional modules and their relationships

The legacy modules diagram targets this challenge. Figure 3 shows the resulting diagram for Conciso. It contains seven functional modules and zero relationships between modules. We present below, how this diagram is obtained and how engineers can take advantage of it and of adjacent tooling to address Challenge 1. Given that Conciso includes .mmb files, we selected the menu-based clustering algorithm option first, from the visualization process wizard.

Once the process is finished, the view is derived; then, engineers have two options to figure out which module contains the deduction form —whose physical name is *CBF55410*— . On the one hand, the first option consists of the following two steps: i) To point each module displayed in the legacy module diagram and ii) To look at the Migration properties view of each module, until finding the deduction form in the *Contents* list. On the other hand, the second option includes the following three steps: i) To open Acceleo Interpreter, ii) To point the root of the Module model and iii) To build an OCL query to ask for the module tat contains the form.

Knowing that the module *General Parameters* contains the deduction form, engineers go back to the diagram to see the module properties: name, size, relationships, etc. The fact that the module *General Parameters* has no ingoing/outgoing relationships, is a signal to engineers that they only have to take care of propagating changes inside the mentioned module (if ever needed). There is no need to worry about other modules when modifying the deduction form. In addition, the diagram shows that there are no relationships between modules, which indicates that Conciso modules are decoupled enough.

We decided to derive another legacy modules diagram for Conciso. This time, we chose the table betweenness clustering algorithm from the visualization process wizard. When comparing the result of this algorithm with the menu-based clustering algorithm result, we observed no correspondences with respect to the number of modules and their content. This fact should not call into question the accuracy of the

Copyright (c) IARIA, 2015.

algorithm. Instead, the reason for this disparity is that the algorithm's derive modules take into account different aspects of software. On the one hand, the menu-based algorithm uses the menu whose items are normally organized in terms of user tasks. As a consequence, the resulting modules diagram maps the final user mental model. On the other hand, the table betweenness algorithm organizes modules taking into account the tables common to different forms. Thus, the resulting modules diagram maps an internal view of the software that is potentially useful to developers. We conclude that in case of having .mmb files, engineers can use the two clustering algorithms results as complementary perspectives. However, in case of lacking .mmb files, the table betweenness is a good starting point for structural understanding.

B. Challenge 2: Relationships between forms and tables

Now, we describe how forms and the table diagram address this challenge. When located in the *General Parameters* module, engineers can navigate the forms and tables diagram until finding the deduction form. Engineers can focus the form following two alternative paths: manual scrolling or an Acceleo query. At the beginning, the diagram shows the relationships between all the forms and tables in the module (i.e. 19 single table, 37 master/detail and 407 PL/SQL relationships), which makes it difficult to focus on the elements that matter. Here is where the filters gain prominence: it is suggested that engineers firstly switch off all filters and, then, progressively turn on each one of them, going from the simplest (i.e., the single table filter) to the more complex (i.e., the PL/SQL filter). Every time a new filter is activated, engineers should analyze the resulting relationships. As an output of filtering, engineers conclude that the deduction form has only a master/detail relationship with two tables, where the master is *CP_MONEY_TYPE* and the detail *CP_CURRENCY_TYPE* (see Figure 4(b)). From this, they obtain knowledge about the tables whose change may impact the deduction form in any way. Subsequently, engineers should complement their knowledge with database information, in order to get a more precise insight about the nature of the impact. A benefit of the diagram, when compared with the manual approach, is that it points out only the tables that are relevant to the form —which would likely speed up the impact study.

C. Challenge 3: Relationships between forms

This paragraph describes how the Forms call dependency diagram targets the third challenge. Once the diagram is generated, engineers can observe that several forms have no dependencies with others. At this point, it is recommended to apply the filter *Single Elements* to leave only the forms that share dependencies. After filtering, engineers obtain the diagram shown in Figure 5. Given the call relationship between form *CBF55400* and the deduction form, engineers can infer that changes in the former will likely impact the latter. Then, engineers need to complement their knowledge with an external source (e.g., the Oracle Forms navigation tree) in order to specify the kind of impact on the related form. Like the Forms and table diagram, the forms call diagram benefit —when compared to a manual approximation— is that it limits the number of forms that have to be inspected during a subsequent impact review.

VIII. Applying approach to illustrating examples

To demonstrate the applicability of our approach, we have obtained visualizations, not only for Conciso but for all the applications mentioned in Section II. Below, we present a table that summarizes what we noticed regarding the resulting diagrams for these applications. Ultimately, we analyze the table data by taking the challenges into account. All tests were executed on a machine with a Windows 7 operative system, Intel Xeon dual core processor and 12 GB of RAM.

TABLE I. Visualization statistics for all applications

Criteria		Conciso	Maestro	Servibanca	Sitri
Clustering Algorithm		Menu -based	Table betweenness	Table betweenness	Menu -based
Modules		7	7	10	69
Module relationships		0	6	5	1
Forms		144	155	83	178
Forms and tables relationships	**Master**	87	52	42	95
	Master Detail	47	43	23	61
	PL/SQL	958	1234	462	1832
Forms relationships		3	154	30	1
Processing time (seconds)		62	83	49	90

- *Modules and relationships between modules:* The module row shows the application size in terms of modules; it ranges from 7 to 69. The latter number indicates not only that Sitri is the largest one from a functional perspective, but also that its menu may be complex due to the large number of options (i.e., 69). In turn, the modules relationships row demonstrates that the modules of Conciso and Sitri have a low coupling. Given that the menu-based clustering was used to derive the "legacy modules diagrams" for Conciso and Sitri, we conclude —from the number of module relationships— that each menu option calls forms that are not called from another menu option. In addition, the modules relationships row shows that Maestro and Servibanca have the highest coupling when compared with the rest of the applications. The rationale behind this result is related to the table betweenness algorithm parameters (i.e., number of database tables in common and number of iterations). It is worth emphasizing that the relationships between modules summarize the relationships between forms contained in different modules. This is the reason why the number of the former relationships is less than the number of the latter relationships.

- *Forms and relationships between forms and tables:* There is a correlation between these rows and the processing time row. The processing time results show that the time spent on the visualization process ranges from 90-50 seconds. The value for each application depends on the application size: the more forms and relationships (either single table, master/detail or PL/SQL), the longer the processing time. For example, Sitri, with the highest processing time, contains much more forms and relationships (i.e., 178 and 1988, respectively) than the rest of the applications.

- *Relationships between forms:* As shown in this row, Conciso and Sitri have few relationships between forms (i.e., from 1 to 3); this occurs because, in these applications, most forms work as independent units accessed through a menu. In contrast, Maestro and Servibanca have much more relationships (i.e., from 30 to 154); the reason is that these applications have no .mmb files. Instead, menus are created manually by adding buttons in .fmb files. As a result, many forms are dependent on these .fmb files.

IX. Conclusion and Future Work

This article proposes a visualization approach for Oracle Forms applications. The diagrams and views have been designed having the ease of modernization in mind. This approach has two main benefits, which were discussed in Section VIII and can be summarized as follows: i) The proposed visualization aids engineers to obtain an understanding of the application. This knowledge can be useful to determine the modernization scope at different abstraction levels: At a high abstraction level, it shows modules that could be potentially impacted by a change made to a form. At a low abstraction level, it points out forms and database tables that are likely affected by the change. ii) The second benefit concerns the productivity of engineers: when compared with the manual inspection of Oracle Forms assets, our visualization approach should reduce the understanding effort in terms of time. This claim will be formally validated by means of a focus group, before project closure.

Also, we outline the four fronts on which we are working on below: i) As was mentioned in Section VIII, the proposed visualization gives an initial knowledge that has to be complemented with information coming from other sources. The navigation from our tool to these sources and vice versa can be tedious, therefore, we are currently working on the integration of the most common source —the Oracle Forms IDE— into our tool; ii) The possibility to reorganize the modules from the diagrams in a way that the new organization is maintained during the migration process; iii) A new functionality that allows engineers to add information to the diagrams —this information could summarize the knowledge they have acquired from the visualizations and from external sources, such as final users—; iv) Finally, a new visualization that looks like a table to display application statistics —such as number of forms, blocks, trigger, etc.— would be desirable. These statistics can be helpful for engineers to estimate modernization costs.

References

[1] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel, "Model-driven engineering for software migration in a large industrial context." in *MoDELS*, ser. Lecture Notes in Computer Science, vol. 4735. Springer, 2007, pp. 482–497.

[2] J. Izquierdo and J. Molina, "An architecture-driven modernization tool for calculating metrics," *Software, IEEE*, vol. 27, no. 4, pp. 37–43, 2010.

[3] M. Riley. (2009) Choosing the right tool. [Online]. Available: http://www.oracle.com/partners/campaign/o49field-084396.html. [Accessed: April, 2015]

[4] T. C. Lethbridge and N. Anquetil, "Advances in software engineering," H. Erdogmus and O. Tanir, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Approaches to Clustering for Program Comprehension and Remodularization, pp. 137–157.

[5] G. Ramalingam and et al., "Semantics-based reverse engineering of object-oriented data models," in *IN PROC. INTL. CONF. ON SOFTWARE ENG.* ACM Press, 2006, pp. 192–201.

[6] R. Bril and et al., "Maintaining a legacy: Towards support at the architectural level," *Journal of Software Maintenance*, vol. 12, no. 3, pp. 143–170, 2000.

[7] O. Sanchez Ramon, J. Sanchez Cuadrado, and J. Garcia Molina, "Model-driven reverse engineering of legacy graphical user interfaces," *Automated Software Engineering*, vol. 21, no. 2, pp. 147–186, 2014.

[8] Composer technologies. Oracle forms to java. [Online]. Available: http://composertechnologies.com/migration-solutions/oracle-forms-to-java/. [Accessed: April, 2015]

[9] VGO Software. Evo. [Online]. Available: http://www.vgosoftware.com/products/evo/walkthrough.php. [Accessed: April, 2015]

[10] Oracle. Jheadstart. [Online]. Available: http://www.oracle.com/technetwork/developer-tools/jheadstart/overview/jhs11-fomrs2adf-overview-130955.pdf. [Accessed: April, 2015]

[11] Pitss. Re-engineering edition-Pitss. [Online]. Available: http://pitss.com/us/products/application-re-engineering-edition/. [Accessed: April, 2015]

[12] Renaps. Ormit. [Online]. Available: http://www.renaps.com/ormit-java-adf.html. [Accessed: April, 2015]

[13] S. Tilley, "Documenting software systems with views vi: Lessons learned from 15 years of research & practice," in *Proceedings of the 27th ACM International Conference on Design of Communication*, ser. SIGDOC '09. New York, NY, USA: ACM, 2009, pp. 239–244.

[14] M. Alalfi, J. Cordy, and T. Dean, "Automated reverse engineering of uml sequence diagrams for dynamic web applications," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, 2009, pp. 287–294.

[15] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana, "Reverse engineering web applications: the ware approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 71–101, 2004.

[16] T. Richner and S. Ducasse, "Recovering high-level views of object-oriented applications from static and dynamic information," in *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, 1999, pp. 13–22.

[17] E. Duffy and B. Malloy, "A language and platform-independent approach for reverse engineering," in *Third ACIS International Conference on Software Engineering Research, Management and Applications, 2005*, 2005, pp. 415–422.

[18] C. Bennett and et al., "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," *J. Softw. Maint. Evol.*, vol. 20, no. 4, pp. 291–315, 2008.

[19] W. Lowe, M. Ericsson, J. Lundberg, T. Panas, and N. Petersson, "Vizzanalyzer - a software comprehension framework," in *Proc. of 3rd Conference on Software Engineering Research and Practise in*, 2003, pp. 127–136.

[20] N. Anquetil and J. Laval, "Legacy software restructuring: Analyzing a concrete case," in *15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 279–286.

[21] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: a clustering tool for the recovery and maintenance of software system structures," in *IEEE International Conference on Software Maintenance*, 1999, pp. 50–59.

[22] M. Girvan and M. E. Newman, "Community structure in social and biological networks." *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[23] Eclipse Community. Eclipse. [Online]. Available: https://eclipse.org/. [Accessed: April, 2015]

[24] Oracle. JDAPI documentation. [Online]. Available: http://www.oracle.com/technetwork/developer-tools/forms/documentation/10g-forms-091309.html. [Accessed: April, 2015]

[25] J. O'Madadhain. JUNG - Java Universal Network/Graph Framework. [Online]. Available: http://jung.sourceforge.net/. [Accessed: April, 2015]

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1