Model-based Assisted Migration of Oracle Forms Applications: The Overall Process in an Industrial Setting

Cristo Rodríguez[a], Kelly Garcés[b], Jordi Cabot[c], Rubby Casallas[b], Fabián Melo[a], Daniel Escobar[a], Alejandro Salamanca[a]

[a] *Asesoftware. Bogotá, Colombia.*
[b] *Universidad de los Andes, School of Engineering, Department of Systems and Computing Engineering. Bogotá, Colombia.*
[c] *ICREA - UOC. Barcelona, Spain.*

Abstract

The migration of software is an economically important niche since the daily operations of many business still rely on systems based on legacy technologies. Migration to more recent technologies comes with major technical and management challenges mainly related to knowledge-loss issues on the specific technology but also on the architecture and design of the legacy. This paper presents a method that addresses the reverse/forward engineering tasks as well as some management challenges such as the planning of the migration based on a better understanding of the system. The process relies on the use of models and transformations to create high-level abstraction views of the legacy. A big challenge with the migration of certain legacy applications is that much of the business logic is inside triggers which are written in the PL/SQL language and scattered across the application layers. Therefore, developers need to know what they need to migrate and where it is. The main contribution of our work is that the abstractions allow developers to understand and plan the migration without the need of having a vast knowledge on the initial technology. Furthermore, our views help developers to decide about the structure and business logic they want to have in the target system. In this paper, we describe the application of the method to the Oracle Forms case study. Nevertheless, the approach can be generalized to other technologies through new bridges that transform from said technologies to the abstract views. The paper ends by reporting lessons learned from our experience.

Keywords: Modernization, Model-Driven Engineering, PL/SQL, Visualizations, Discovery, Code Pattern

Email addresses: crodriguez@asesoftware.com (Cristo Rodríguez), kj.garces971@uniandes.edu.co (Kelly Garcés), jordi.cabot@icrea.cat (Jordi Cabot), rcasalla@uniandes.edu.co (Rubby Casallas), fmelo@asesoftware.com (Fabián Melo), descobar@asesoftware.com (Daniel Escobar), asalaman@asesoftware.com (Alejandro Salamanca)

1. Introduction

Most organizations wish to migrate their legacy applications to new architectures and technologies, pressured by factors such as high maintenance costs, technical debt, inability to cope with incoming requirements and regulations, incompatibility with other systems, ceasing support by vendors, and lack of experts on out-of-date programming languages and technologies.

A migration project involves two main phases: i) reverse engineering; and ii) forward engineering [1]. Reverse engineering aims at understanding the current system through representations at a higher-abstraction level (e.g., different kinds of software models) compared to that of code. The reverse phase may include restructuring of the source system to favor quality attributes in the target application. Forward engineering is the process of transforming those high level representations into the physical implementation of an application [2].

There are many challenges in a migration project: insufficient documentation, lack of knowledge about the legacy technology, unclear application's architecture and software structure [3], and a big deal of uncertainty about how to estimate and plan the project [4].

We have been working, for some years, in collaboration with industry partners to carry out migration projects that target diverse technologies [5, 6] such as Oracle Forms [7, 8, 9], Java [10], and Ruby on Rails [11].

The migration of Oracle Forms (OF) is our migration project most advanced and mature. Forms applications are present in many sectors. Results of a tool usage survey [12], carried out by the Oracle User Group Community Focused On Education (ODTUG) in 2009, indicate that 40% of 581 respondents (application developers) use Oracle Forms.

We undertake a long collaboration with Asesoftware, a Colombian software company that offers migration services to its clients who desire to translate OF source code to other technologies such as Java or .Net. This joint work dates from 2014. As a result of this work, we have managed to incrementally propose a technology-independent migration method supported by a commercial product called SMoT (Smart Migration Tool). SMoT is based on Model-Driven Engineering (MDE) techniques [13]. SMoT was designed to tackle the challenges mentioned above. One cornerstone component in our solution are the languages (referred to as *metamodels*) describing 4GL technologies. These languages, together with parsers, editors and model transformations, strives towards technology independence. One important contribution is that people who do not known very well Oracle Forms technology can work using the high level abstraction views not only to understand the architecture but also to modify and reorganize, at that level, the initial application to favor quality attributes in the target application. Once decision making is performed, SMoT is able to produce an application written in the target language. It is worth noting that SMoT addresses the code translation of the application *structure* that refers to the graphical interface arrangement (i.e., basically through forms) and how the interface is mapped to database tables to perform basic CRUD (Create/Read/Update/Delete) functionality.

Besides the *structure*, a characteristic of these applications is that most of the *business logic* –that goes beyond the CRUD functionality – is inside small programs written in

PL/SQL which are triggered by events.

When reviewing the literature, we found that the PL/SQL issue has been addressed in two ways. On the one side, some works [14, 15, 16] reverse engineer large legacy PL/SQL applications to achieve program comprehension but do not address the translation of such code. On the other side, other works [17, 18, 19] deal with the reverse and forward engineering tasks involved in the migration of PL/SQL code which is embedded in Oracle Forms applications. They focus on modeling PL/SQL language statements at a syntactic level.

Like the first group of referenced works [14, 15, 16], we aim at reverse engineering the PL/SQL code with a program understanding intend. Thus, a fully automated translation of the application *business logic* expressed in the PL/SQL code is out of the paper scope. For leveraging program comprehension, SMoT executes an algorithm that matches the PL/SQL blocks against a catalog of *code patterns*. A code pattern is a set of statements that frequently appear together in the code and that implement a specific business logic validation or functionality. After that, SMoT displays the PL/SQL code –classified according to functional purpose– in a view for the analysts to decide which triggers worth to be migrated and which ones should be skipped (for example, because of technology obsolescence). Finally, in the generation step, the approach embeds the PL/SQL snippets –marked as "to migrate"– in strategical locations of the target code for developers to manually translate them.

The difference with respect to previous works is that our approach goes beyond the syntactic level: it specifically targets the semantics of this domain by first grouping PL/SQL statements that are typically used together. Furthermore, we aim to discover business-level functionality regardless of the number and complexity of the statements required to perform it.

In addition, to tackle the challenges related to project management, on top of the legacy/intermediate models, SMoT has functional features to help developers to estimate, plan, and track the project progress.

In a nutshell, the contributions of this work are: i) a platform-independent migration method that goes beyond the technical translation from source to target, as it includes managerial steps such as planning and project progress tracking. It is worth noting that said translation tackles the applications *structure*; ii) a technical contribution regarding the comprehension of PL/SQL code; and iii) lessons learned in applying our method to an industrial setting.

Technical and managerial concerns are addressed by using views (or editors) that show the information available in three different models referred to as *legacy model* [7], *intermediate model* [8, 9], and *code patterns model*. The migration method itself is separated in two processes: the creation of the migration tool (named "Base migration process") and the application of this tool in a particular case (named "Migration application process").

The paper is structured as follows: Section 2 describes the two aforementioned processes in an agnostic manner for them to be replicated in other contexts (for example, if the source/target technologies are changed). Section 3 introduces the OF industrial setting, which was performed in collaboration with Asesoftware. Section 4 describes the application of the "Base migration process" to the case study. Afterward, Section 5 presents the notion of PL/SQL code pattern and explains how these patterns are matched against the application

code. All these sections report design decisions that could be useful to practitioners that face similar challenges and wonder which are the alternatives (including their benefits and drawbacks) when carrying out model-based migration. Section 6 discusses the results of applying the "Migration application process" to a project carried out in a banking company. This evidence shows improvements in developer's productivity and code quality when using our SMoT tool. In addition, developers expressed that the new (semi)automated process is less complex than the one they manually carried out before. Section 7 exposes some lessons learned. These lessons are categorized into two groups: use of MDE techniques and code patterns. Section 8 addresses the literature review and Section 9 presents our conclusions and outlines future work.

2. Model-based Assisted Migration Processes

Even though the source and target technologies of the case study are particular (Oracle Forms and Java/.Net), we have designed the migration approach to be Technology-independent. For each new source or target technology we have to develop some assets (referred to as the *Base migration process*) that are then reused for every project that conforms to such technologies (this is called *Migration application process*).

Fig. 1 (which uses a UML activity diagram notation) shows the two processes. Ovals represent activities and arrows represent transitions between activities. The horizontal synchronization bars represent that completion of the previous activities before starting new ones is required. Processes are detailed in the remaining of this section.

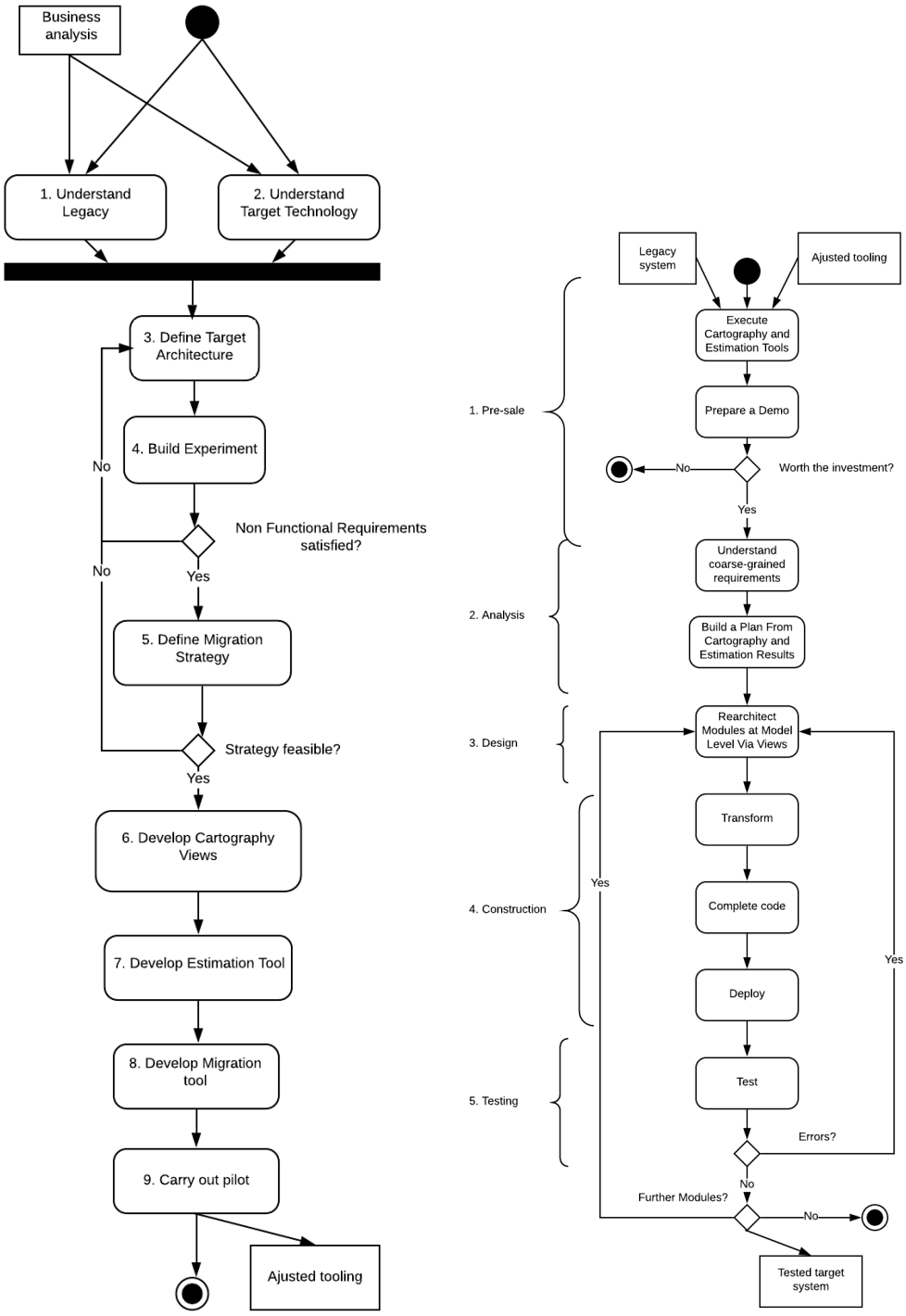
2.1. Base migration process

This process is carried out if the *business analysis* shows that the migration is financially viable. The analysis studies the business value that the migration from a particular source to a target technology would bring to the company. The study includes: i) identification of stakeholders and migration goals; ii) evaluation of similar migration efforts of competitors; and iii) definition of plan and cost estimation for the base migration process.

Therefore, the *base migration process* starts with the business analysis and its end state is the tooling that assists users (such as analysts, architects, developers) in migration projects. This process should be led by a team consisting of Model-Driven Reverse Engineering (MDRE) experts, legacy experts, and software architects. This process has the following steps (see Fig. 1 a):

Step 1: Understand legacy. Its purpose is to achieve a better knowledge of the technologies used to construct the legacy systems. This step is often complex because it involves the comprehension of languages, libraries, operating systems, tools (e.g., Integrated Development Environments), component systems, database systems, etc.

Step 2: Understand target technology. This step consists in studying how the target technology contributes to the fulfillment of quality attributes of interest. Often, this understanding is more straightforward compared with that of the previous step because most of the team members likely have more experience using modern technologies than legacy.



(a) Base migration process. (b) Migration application process.

Figure 1. Processes description.

Step 3: Define target architecture. This step encompasses the following sub-steps: i) prioritization of Non-Functional Requirements (NFRs); ii) selection of architecture styles, tactics and patterns to address the NFRs; and iii) design of architecture diagrams (e.g., components, deployment, etc.) that reflect decision making.

Step 4: Build an experiment. In this step, the team constructs an experiment (that is, software + tests) to verify that the proposed architecture satisfies the NFRs. If so, the team members continue the process. Otherwise, they continue iterating until a compromise between technical and business constraints is achieved.

Step 5: Define migration strategy. According to Seacord et al. [20], there are different kinds of software migration; for example, revamping and source code translation. Revamping refers to the replacement of the UI of a system, whereas source code translation refers to the conversion of an application from a programming language to a different one.

Once the team members establish the migration strategy, they decide the scope by answering the following questions: i) which constructs of the legacy are kept as-is?; ii) which constructs of the legacy are worth migrating to the target?; iii) which are the mappings between these constructs and the target technology?; iv) which parts of the legacy will be migrated (semi)automatically?; v) which parts of the legacy will be migrated by hand?; vi) how to properly indicate to developers the location where code completion is needed?; and vii) what kind of mechanism will preserve automatically generated code from mistakes manually introduced by developers?

Answers to these questions serve to specify the requirements for cartography, planning and code translation.

Step 6: Develop cartography views. This step consists of building abstractions on the legacy applications (architecture views). 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.

Step 7: Develop estimation tool. This step entails the construction of a tool that helps the development team to estimate the migration cost in an accurate manner. A variety of estimation methods can be found in the literature [21]; for example: expert judgment, tasks breakdown, parametric methods, functional points, use case points, etc. The chosen method will depend on the availability of experts and legacy artifacts (mainly code and documentation).

Step 8: Develop migration tool. This step includes the construction of a tool that produces the target application from the legacy, bearing in mind the migration scope. Model-driven migration normally requires complex transformation chains that take as input the legacy code, translate it into several intermediate models until the target code is produced. If the migration process follows a white-box approach, then it is necessary to provide with means for the users to make architecture decisions on top of the intermediate models.

Step 9: Carry out the pilot. The purposes of this step are to: i) fix bugs of the tools; ii) measure the gains in productivity and quality to adjust the business model; iii) adjust the instantiation process because the legacy may require preparation prior to migration or the target application require further manual completion after the migration. Even though the validation of the migration tooling is performed at the end, tests and intermediate meetings to show the progress have to be scheduled in the base migration plan.

2.2. Migration application process

The process starts with the legacy system of a particular client. Its end state is the target application. The development team (including analysts, architects, developers and testers) is supported by the migration tooling built in the base migration process. The instantiation consists of four phases, each of which comprehends one or more steps. A description is given below (see Fig. 1 b):

Step 1: Pre-sale. The team takes advantage of the cartography, estimation and migration tools to build a demo that shows the capabilities of the approach. Whilst the cartography and estimation are performed on the entire legacy application, the migration is carried out on a particular artifact often of low complexity to speed up the pre-sale. The process continues if the client decides that the investment is worth it.

Step 2: Analysis. The team organizes meetings with the client where the latter transfers their knowledge about the application functionalities. This knowledge serves to understand if the migration tool requires any adaptation. The adaptation could be motivated by two aspects: i) lack of information in the legacy that may hamper a straightforward migration; or ii) requirements of small changes in the target architecture (for example, application look-and-feel and security mechanism). At this stage, it is important that the team informs the client about the adaptation cost for her to decide if the project goes or does not go. If it continues, the team defines an incremental plan of migration based on the client's priorities.

Step 3: Global/detailed design. Most of design decisions were made in the base migration process. As a consequence, at this stage, the remaining decisions aim at re-architecting the application by using the configuration features provided by the development team. There are decisions that are global; for example, restructuring of a menu to help users find the options they are looking for. But also, there are fine-grained decisions; for instance, to introduce/delete/modify graphical components to display the information that actually matters in the target application. Whilst global decisions are made before starting the transformation of any particular module, detailed decisions are made on the constituent elements of a pre-established module.

Steps 4 and 5: Construction and Testing. The development team carries out these two phases in an incremental manner following the migration plan. For each iteration, the team chooses a set of functional modules, performs a detailed design, executes the transformation from legacy to target, completes the code, deploys and tests. If errors are detected, the team goes back either to the design step or to the manual migration step. Otherwise, the team proceeds with the modules indicated in the plan until reaching the end.

3. Case study overview

Asesoftware[1] is a Colombian software company with 250 developers. It has a longstanding experience in development and maintenance of OF systems for banking and insurance sectors. Asesoftware and the Universidad de los Andes carried out the "Forms migration" project, which was partially funded by a grant from the Colombian government. The purpose of this grant was to contribute to the improvement of the communication between local researchers and companies, in order to facilitate the development of new technologies, products and processes, which would in turn result in better competitiveness for the productive sector.

The "Forms Migration" project arose out of the need to provide a comprehensible and usable solution to some reoccurring problems faced by Asesoftware in its migration projects. This company offers services amongst which are the migration of Oracle Forms applications to Java or .Net. In this particular context, the project's main output was a semi-automated migration process [7][8], which was leveraged by MDE technologies that are within the domain of expertise of the involved researchers. Said process and supporting tools tackled the following challenges: 1) decreasing the workload required to understand the structure of Oracle Forms applications; 2) transforming Oracle Forms to Java or .Net; and 3) generating unit tests. The transformation scope includes: 2.a) the graphical interface of the forms (except the layout); 2.b) the generation of typical CRUD (Create/Read/Update/Delete) operations for those forms.

To carry out the project We formed a joint team with ten people from Asesoftware (including the executive director, a project manager, a senior software architect, a legacy expert, and five junior developers) and four people from the University (i.e., three professors and an engineer). In next paragraphs we describe how was the collaboration during the base migration process and the application of the migration tooling to legacy of Asesoftware clients.

In the base migration process the company performed the business analysis before contacting the University and they decided the source/target technologies based on their clients' requests and market tendencies. The first six months of the project, the University members went through the learning curve of the legacy technology and were accompanied by the Asesoftware legacy expert. The team defined what was worthwhile to be migrated to target applications. For example, keyboard shortcuts to trigger functionality were determined to be skipped during the migration process since in the target technology (i.e., Web) the functionality is mainly launched through the mouse. Based on this decision making, the University team proposed an initial version of the architecture further refined by the Asesoftware architect. After that, the team mocked the cartography views bearing in mind the nature of the legacy (e.g., files naming convention, structure, dependencies, etc.) and the University members built the cartography tool. All this collaboration promoted knowledge transfer between Asesoftware and the University, so much so that Asesoftware members were fully autonomous to develop the estimation tool by using MDE technologies. In the

[1] www.asesoftware.com

last six months of the project, the University members together with an Asesoftware junior programmer developed the migration tool. The end of the base migration process, i.e., the pilot, was mainly leaded by Asesoftware. The University members (in particular the engineer) basically contributed in bug fixing and training of the subjects who participated in said pilot.

The application of the migration tooling within clients was also mainly leaded by Asesoftware. The University members were basically consulted about adjustments in the process.

Previous paragraphs describe the experience during the financed project. After that, we carried out one academic project with Asesoftware employees who were enrolled in the Software Engineering master of the University. This project allowed us to tackle the challenge of PL/SQL migration which was skipped in the first work.

Based on a long experience of over 20 years in the maintenance of Oracle Forms applications, Asesoftware engineers estimate that the effort invested in these tasks is about 36% of the total project effort, that is why PL/SQL migration was the next step to tackle during the collaboration. As a technical contribution, Section 5 presents an approach to decrease the effort of discovery and smooth the transformation by embedding the original PL/SQL in strategical locations of the target code. With this solution, developers avoid switching between legacy/target IDEs and focus on the PL/SQL code migration.

4. Base migration process for case study

The base migration process for the Oracle Forms case study took one year. We finished the process on time although at one point in time we were behind the initially agreed schedule in the steps referred to as "understand legacy" and "build cartography tool" in Fig. 1a, for two reasons: i) limited availability of legacy experts; and ii) long learning curve of the framework (i.e., Sirius[2]) on top of which the views are built.

Fig. 2 presents the overall tooling produced in the base migration process. Note that the figure highlights the models, operations and outputs of each tool, i.e., cartography tool, estimation tool, and migration tool which are connected to steps 6, 7, and 8 of the base migration process illustrated in Fig. 1a. The most relevant models in the tooling are: i) *Legacy model*; ii) *Intermediate model*; and iii) *Code pattern model*. The legacy model has a one-to-one correspondence with the code. The intermediate model is a Platform-Independent representation (PIM) of the application. The code pattern model collects all the application triggers and categorizes them into the typologies of patterns of a catalog (if possible).

It is worth noting that we devoted considerable amount of time to the intermediate model design. We discussed whether or not using an existing PIM. For example, the Knowledge Discovery Metamodel (KDM) is a standard PIM to represent and manage knowledge throughout all the modernization stages. We used this PIM in previous research [10] and in the context of an MDE course for master students. We have observed two disadvantages: i) it is tough for developers to appropriate the KDM concepts because the metamodel is quite large; ii) some concepts are too abstract, leading developers to different comprehension and

✎ Remark 1

[2]https://www.eclipse.org/sirius/

application of those concepts, creating confusion in the team. For example, they tend to map the same legacy concept to different KDM concepts that look similar at first glance. We are aware of the importance of standardization. However, given our interest in smoothing the appropriation of the intermediate model among practitioners (i.e., a better learning curve), who are not necessarily MDE experts, we designed an Ad-hoc PIM that has less concepts than KDM and focuses on the domain of 4GL technologies.

Returning to the tooling description (see Fig. 2), it has the following elements:

- A parser written in Java creates the legacy model from the code.
- An Xtext parser creates an abstract syntax model (ASM) from the PL/SQL code.
- A model-to-text (m2t) transformation generates a cost estimation spreadsheet from the legacy model.
- An editor, that works on top of the legacy model, graphically displays the legacy structural parts and their relationships.
- A model-to-model (m2m) transformation generates the code pattern model from the ASM.
- An m2m transformation generates the intermediate model from the legacy model.
- An editor, that works on top of the code patterns model, shows the application triggers and allows developers to decide which triggers have to be migrated and skipped.
- An editor shows the elements of the intermediate model and allows developers to make configurations.
- An m2m transformation merges the new version of the intermediate (the one resulting from developers' decision making) and code patterns models.
- An m2t transformation generates the target application for the developers to manually migrate the PL/SQL code (taking into account the comments injected by the tooling) and executes the tests.

Note that the tooling follows the three stages of model-based reengineering: reverse engineering, restructuring, and forward engineering. The reverse engineering stage covers the injection of source code into the legacy and code pattern models. In turn, the restructuring stage covers the decisions that developers can make on top of the intermediate and code patterns models. Finally, the forward engineering stage takes these models as input and produces the target code.

In this section, we briefly present how each step of the process was carried out for the case study. We spell out implementation details in the steps where tooling is used. Next section gives an in-depth description of the technical novelties introduced in this paper, which are linked to the base migration process steps referred to as 6 and 8 in Fig. 1a.

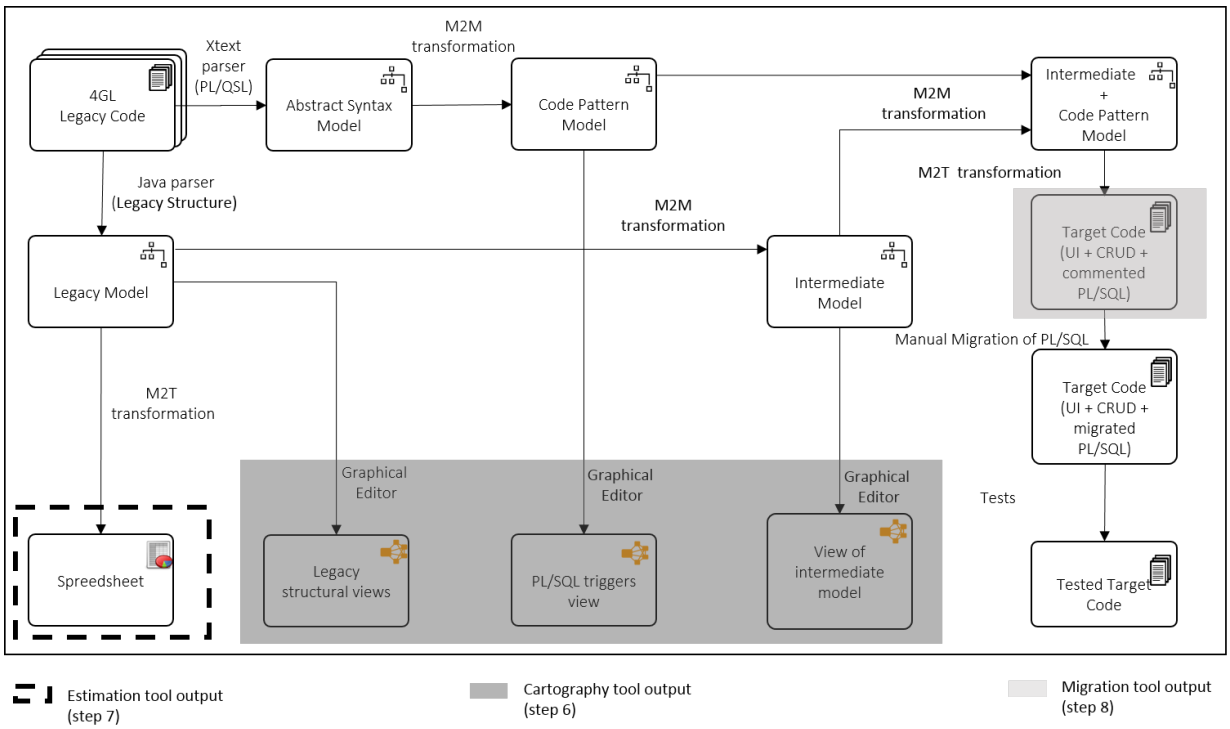


Figure 2. Models, operations and outputs of the tools developed in steps 6-8 of the base migration process

4.1. Step 1: Understand legacy

Oracle Forms appeared towards the end of the 1980s as a programming language and development tool for creating desktop applications that interact with Oracle databases. Oracle Forms provides a rapid application development environment plus a runtime engine that allow programmers to create database-centric functionality, thus minimizing the need for programming common operations such as transaction management and coding of CRUD operations.

Oracle Forms also allows developers to include PL/SQL code in the applications in order to enrich its functionality beyond CRUD standard logic. PL/SQL is a procedural programming language and a proprietary development by Oracle. The PL/SQL code can be executed by the database management system (in the way of database triggers or stored procedures) or directly by the Oracle Forms environment (as code excerpts embedded in the Form trigger events). Our proposal focuses on the PL/SQL embedded in those forms.

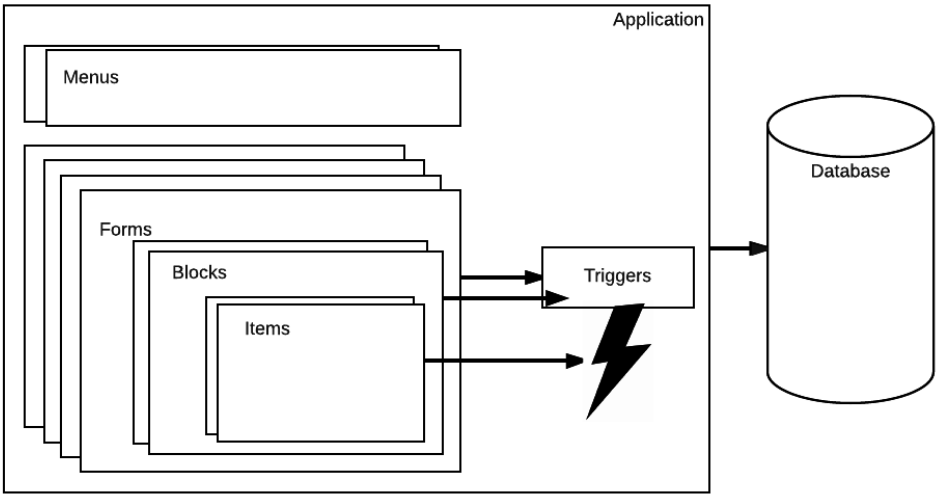


Figure 3. Simplified version of the Oracle Forms architecture. Taken from Garcés et al. [9].

Figure 3 illustrates the main concepts of Oracle Forms applications and a description of such concepts follows:

- *Form*: A form is a set of interface objects (check box, lists, grids,...) grouped in blocks plus code excerpts.
- *Blocks*: Represent logical containers for grouping related items into a function unit to store, display and manipulate records of database tables. Typically, programmers configure blocks depending on the number of tables from which they want to manipulate the form (one block per table accessed in the form). For instance, the way to display a single database table in a form is to create a block as a master form. To display two tables that share a master-detail relationship (i.e., "One to Many" relationship) you

would configure and link two blocks. As expected, then, Oracle Forms would guarantee 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 form.

- *Item*: User interface element that represents a field/column, and allows the user to display and modify its value. Oracle Forms offers several types of items: Text, Check Box, Radio Groups, and List, among others. Some items have a PL/SQL logic that "looks up" data when the application runs. Item might be referred to as Field in the remaining of the paper.

- *Forms Triggers*: Developers can define adhoc application logic as part of the form, block and item definitions by adding Form triggers, which are routines of PL/SQL code fired in response to user actions, validations and record management events. Examples of triggers are: *When-Checkbox-Changed*, *When-Validate-Record*, and *When-Remove-Record*. The code of a Forms trigger is different than that of a database trigger. In this paper, we use the term Trigger to refer to the former. We show an example in Listing 1. This code ensures the consistency of a foreign key constraint in a Master/Detail form, avoiding orphaned detail records when a user is trying to delete a master record. A query over the BRANCH table (this is the detail table in a master-detail relationship) is declared (line 4), filtered by COMP_CODE. If the query retrieves at least one record, an exception is fired and the operation is aborted (lines 10 to 14).

Snippet 1. Delete validation on Master-Detail relationships

```
1 declare
2  cursor primary_cur is
3   select 'x'
4   from BRANCH where COMP_CODE = :BLK_COMPANY.CODE;

6  primary_dummy  char(1);
7 begin
8    open primary_cur;
9    fetch primary_cur into primary_dummy;
10   if ( primary_cur%found ) then
11     message('Cannot␣delete␣master␣record');
12     close primary_cur;
13     raise form_trigger_failure;
14   end if;
15   close primary_cur;
16 end;
```

4.2. Steps 2-4: Understand target technology, Define architecture and Build experiment

Asesoftware decided on a multi-tiered architecture for the "Forms Modernization" project by taking into account local market needs. The migration takes Oracle Forms code as input and generates a target application that implements this architecture in Java or .Net. The aforementioned multi-tiered architecture and its corresponding technologies is detailed in [8].

The team from the university built a prototype satisfying the aforementioned architecture. The prototype underwent functional testing. Testers suggested to change the target UI look-and-feel to bear a high degree of likeness to the legacy; for example, keeping master and detail records in the same screen. This was necessary to ensure that final users' experience was affected as little as possible.

4.3. Step 5: Define migration strategy

With respect to the kind of migration we decided to perform *source code translation*. The reason for this was that Asesoftware wanted to reduce vendor lock-in as much as possible.

Connected to the kind of migration, we decided between Black-Box and White-Box migration styles. Black-Box approaches generate applications that conform to a default architecture instead of letting the user decide about some aspects. As a result, users can only specify and implement their decisions over the generated code. In addition, means to see the transformation progress are uncommon in most of the related work. To tackle these limitations, we have contributed a White-Box approach with different models and editors that allow developers to make decisions at model level (without a mandatory knowledge of the legacy) in the early steps of the translation and see the project progress.

Furthermore, We decided which aspects of the legacy to migrate and which ones to leave out. We tried to find a balance between adding value to the company and finishing on time. Actually, we were accountable to the Colombian government and had to comply with a strict schedule.

First, we focused on the UI and logic layers leaving out the database. In the business analysis step, Asesoftware's clients (mostly from the financial and the insurance sectors) claimed to want to minimize the risk of leaking or tampering database information.

Second, we automated the migrations of *master* and *master/detail* forms because of their widespread use in enterprise applications. Third, we decided to generate the corresponding graphical elements of a given form but leaving its layout out of the scope. Second and third decisions brought kind of architectural debt that the team had to pay during the migration project within a particular Asesoftware's client.

With respect to the third decision, the team had to devote a considerable amount of the time to implement an algorithm to discover the concrete legacy layout. The rationale was that Asesoftware's clients wanted to reduce as much as possible the time that final users spend exploring and learning the new user interface. This fact reaffirms the importance of the "pre-sale" and "analysis" steps where the migration team has to identify if the tool requires any adaptation given the client's particular needs.

We concluded that the automation of the PL/SQL translation has a high effort that would not be worthwhile due to the variety of patterns that one can find across applications developed by different teams. That is why we decided that the actual transformation of PL/SQL code to the target language is the responsibility of the developer.

In this sense, we proposed a middle-ground solution to support developers in this task independently of the patterns used. The support is in two senses: i) developers can select which PL/SQL blocks to migrate on top of a view that harvests and categorizes this code; ii)

the blocks selected by the developers are embedded into appropriate locations of the target application as commented code, thus avoiding switching between the legacy and target IDEs.

4.4. Step 6: Develop cartography views

At this step it is important to determine whether to use standard or domain specific views. 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 [22][23][24][25] is the Unified Modeling Language (UML). 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 [26].

We decided to build domain specific views taking into account the aforementioned rationale. Another reason was that OF application code is not as structured as object-oriented one. Clients basically provide a folder that contains forms on the root, with no subfolders nor documentation. Each form has a name that is the concatenation of a prefix 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.

Thus, the proposed views are part of the cartography tool and allow the migration team to get insights about the legacy. There are two categories of views at this stage: i) structural views of legacy; and ii) PL/SQL view.

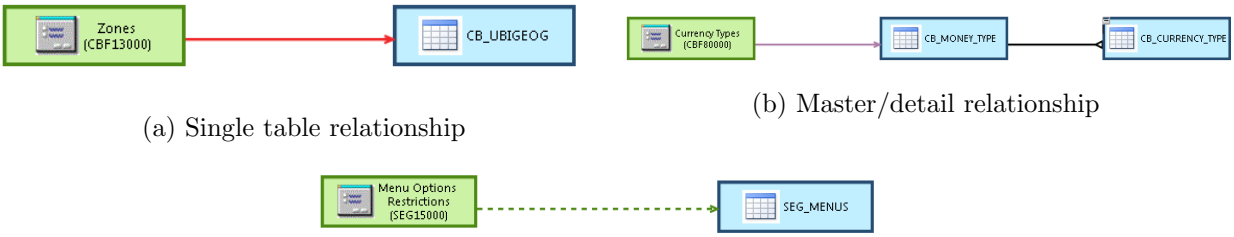
4.4.1. Structural views of legacy

These views allow the development team to understand the structural organization of the legacy; that is, i) subsystems; ii) what elements constitute these subsystems (e.g., forms, database tables); iii) and the relationships between these elements. We designed three views for the case study: i) Functional modules view; ii) Forms and tables view; and iii) Forms call dependency view. As shown in Fig. 2, these views are built on top of the legacy model. A Java parser receives the legacy application as a set of Oracle Forms files (.fmb) and produces the legacy model. Readers interested in further technical details about the view generation can review our previous work [7].

Figure 4 illustrates the functional modules view. An orange circle represents a Module. The circle label is the module name, which can be changed by the user for a more meaningful name. The circle size is proportional to the number of form elements contained in the module. In this concrete example, there are no relationships between modules; that is why arrows do not appear.

It is worth mentioning that the Functional modules view is useful for planning because it shows how a legacy system is organized in terms of modules or subsystems and what the relationships between the modules of a system are. Knowing the modules helps engineers to plan the order in which the parts of a system will be modernized. Furthermore, engineers

Figure 4. Legacy modules view [7]



(a) Single table relationship

(b) Master/detail relationship

(c) Call relationship

Figure 5. Excerpt of the Forms and tables view [7]

can use the relationships as indications about potential impact on forms belonging to other modules, as a consequence of migrating a given form.

Fig. 5 illustrates the Forms and Tables view. 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).

The relationships between forms and tables are important because they suggest to engineers how database tables feed UI elements and are operated in embedded PL/SQL code.

During step 6 we discussed how to favor the reuse of such visualizations. If the development team wants to recycle views for different source/target technologies, then they need to specify the views on top of a unique PIM. However, it is possible that some views could not be specified because the PIM lacks source/target information. Indeed, the generalization of concepts from different domains in a unique PIM likely mean sacrificing particularities of such domains.

We actually experienced this in the project. The editors available on top of the intermediate and code patterns models allow developers to make decisions on the target architecture. These views are reusable among different source/target technologies. However, we implemented the structural views on top of the legacy model because the latter has syntactic sugar (e.g., the type of relationships established between Forms elements) that is missed in the PIM.

4.4.2. PL/SQL view

This view allows the legacy expert to plan which triggers are worth migrating to the target application. The expert displays and navigates the Code Pattern Model by using the view. The purpose is to mark each trigger either with the label "to migrate" or "to skip". Based on this, the generation step either embeds or does not embed the source PL/SQL code in the target code for the developer to manually transform it.

The team evaluated two alternatives for the model on top of which this view was built. Because the legacy code embeds in the same place both structural elements and the PL/SQL blocks devoted to the interaction with those elements, one alternative would have been to represent both concerns in the intermediate model.

Instead, we decided to follow an alternative path in which these two concerns are represented in separated models, i.e., the intermediate model and the code pattern model, respectively. The rationale was to reuse the code pattern model independently of the particular legacy technology. Whilst the structural elements among different legacy technologies may have few variations, the use of PL/SQL to enrich functionality beyond CRUD standard logic is standard among different 4GL technologies.

Figure 6 shows how the view looks like. Note that triggers appear classified by category. The expert can see the source PL/SQL code by clicking on each occurrence. A given trigger can appear classified in more than one typology. The reason for this is that such a trigger may consist of several blocks, each of them matching a different pattern. Section 5.1 specifies the catalog of code patterns that have been extracted from PL/SQL code. Section 5 explains the technical details of the Code Pattern Model production.

4.5. Step 7: Develop estimation tool

Prior to developing the tool, we had to decide what estimation strategy will underlay the tool. Asesoftware chose Function Points (FP) as a means to measure the size of legacy systems and used it to estimate the migration development effort. The company decided to base the migration size estimation on FP because: i) they have been using the method during a substantial period (more than 20 years) and a large project base; and ii) they have managed to recollect historical metrics of the projects effort based on function points as unit-of-work.

After decision making, we implemented the estimation tool consisting of a transformation and an Excel template. The transformation navigates the legacy model and produces csv files whose rows represent the form elements and its corresponding function points. In turn, the total number of FPs and the effort is calculated by the template from the csv files. Finally,

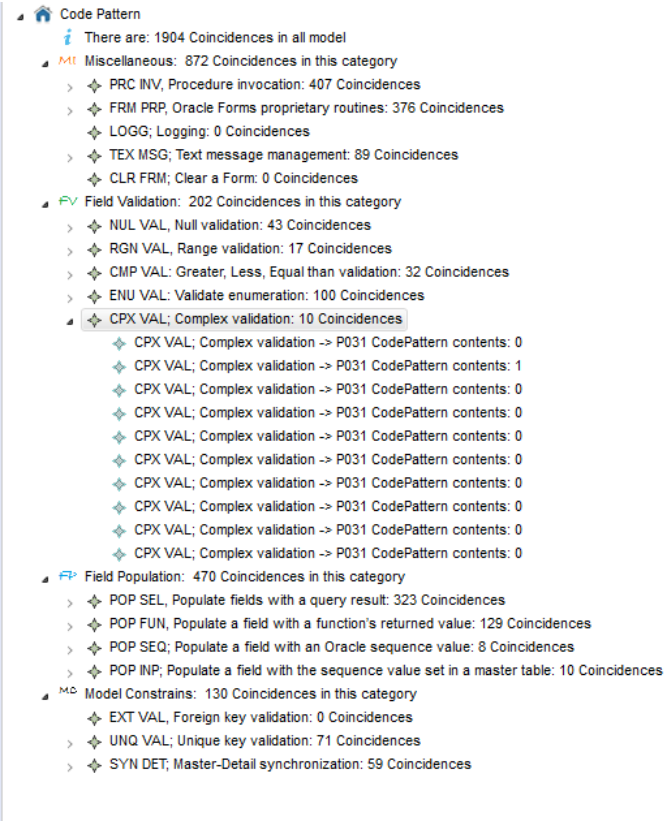


Figure 6. Screenshot of the view of categorized triggers

an expert reviews and cleans the data to eliminate items that might (under/over)charge the estimation. The details of this step are out of the scope of the paper.

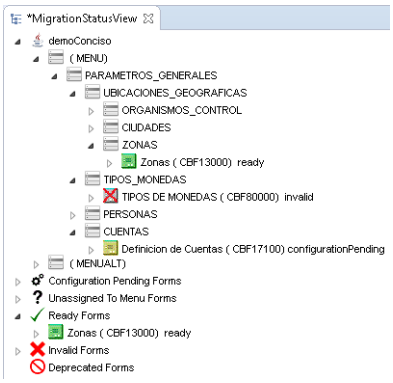
4.6. Step 8: Develop migration tool

The migration tool takes as input the Code Pattern Model produced in the cartography, merges such a model with the intermediate model, and generates the target code (including CRUD functionality, User Interface, and commented PL/SQL). In the last step, developers manually migrate the PL/SQL code. Our approach smooths the migration since developers do not waste time figuring out, from the target code, whether an excerpt has to be translated. Instead, the decision has been already made by the legacy expert at design level via the Code Pattern Model and the PL/SQL view.

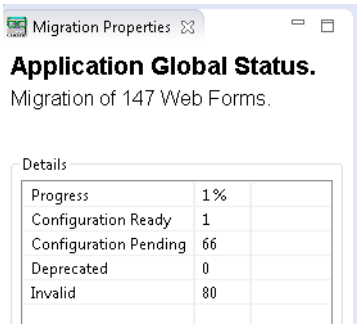
We implemented the migration tool as a model transformation chain consisting of two model-to-model (m2m) transformations and a model-to-text (m2t) transformation (see Fig. 2). One of the m2m transformations translates from legacy to intermediate model and the other merges the code pattern elements with affected intermediate model elements. The m2t transformation needs this association to know where each code excerpt needs to be executed. The association is represented through a relation between the "Code Pattern" and "Event" concepts in Figure 8. This step is straightforward since the intermediate model keeps track of

the name of triggers linked to user interface components therefore, during the merge we just need to join both models and merge the triggers with their patterns by name matching.

Prior to migration, developers can make decisions on the target architecture by using an editor that displays the intermediate model. Below, we present this editor's main functionalities and spell out how they support the configuration of the target. For illustration purposes, we include some screenshots (see Figures 7a and 7b) that show how the intermediate model is displayed by the editor.



(a) Menu structure and screen classification.



(b) Migration project progress view.

Figure 7. Functionalities of the Intermediate Metamodel Editor [9]

4.6.1. Menu structure definition

If the legacy application owns *.mmb* files, then the editor displays its menu items. For example, as shown in Figure 7a, the application menu items are: *ZONAS, CUENTAS*, etc. In the absence of *.mmb* files, this section is going to appear empty, in which case, the developer can create menu items and associate screens to them from scratch. In any case, the developer can restructure the menu by using drag and drop tools. This functionality targets usability.

4.6.2. Screen classification

In the second step, an algorithm assigns an initial migration status to each screen according to the information present in the intermediate model. Based on this information, the editor is able to classify the screens into five categories that allow the developer to easily figure out the screens that are included in a given configuration/validity status (see Figure 7a): 1) *Configuration pending screens*: lists the screens that lack a basic configuration, that is, name, primary/foreign keys for associated entities, and roles. 2) *Unassigned to menu screens*: groups the screens that are not associated to any menu item. It is important to warn the developer about this fact because, otherwise, the final user would not be able to reach the functionalities of these screens from the modern graphical interface. 3) *Ready screens*: presents the screens whose basic configuration is ready. 4) *Deprecated screens*: groups the screens marked as deprecated by the developer. These screens will not be transformed. 5) *Invalid screens*: lists the screens that are out of the modernization scope,

that is, the screens that are not master or master/detail. The second and fourth categories point to maintainability. The third category targets the project progress concern. Finally, the fifth category enforces compliance to the modernization scope.

4.6.3. Application configuration status view

The developer has access to this view from the root element of the editor. The view (see Figure 7b) summarizes the configuration progress of the application screens, the number of screens in the status ready, configuration pending, deprecated, and invalid. Therefore, this view targets the project progress concern. To obtain the percentage, we divide the number of ready screens with the result of subtracting the number of deprecated screens from the total number of screens.

4.7. Step 9: Carry out a pilot study

We carried out a pilot study whose main purpose was to compare the productivity of developers who manually transformed legacy with that of developers who used SMoT.

The subjects were 4 developers from Asesoftware. These developers were divided into 2 teams, namely "Manual Team" and "SMoT Team". The sample comprised 4 men, of whom 50% were Senior Java developers (at least 5 years of professional experience) and 50% Junior Java developers (between 0-2 years of professional experience). Each team comprised 1 senior and 1 junior developer. By the time the experiment took place, the subjects from both teams received training in the relevant technologies. Such a training consisted of 8 hours of training in Oracle Forms, 4 hours in the target architecture and 14 hours in the white-box method. Two forms of different complexity were transformed by each team.

Developers were scheduled to work on these two forms during three weeks. We strictly defined that both teams had to keep the target architecture. This way, it was possible to standardize to a certain point the target code that had to be developed and facilitate its measurement. In order to convert the forms, the Manual Team followed Asesoftware's initial method, which consisted of a code-centric transformation from scratch. The other team, in turn, followed the white-box transformation process assisted by SMoT. In a time tracking tool, both teams reported the effort of carrying out each method. In addition, defects were identified through functional tests.

The pilot showed that developers are significantly more productive (approximately 40%) and the quality of the new code is significantly higher (environ 61%) when using SMoT than when using the manual migration approach. See [9] for further details on the pilot.

5. Discovery of PL/SQL Patterns Occurrences

The PL/SQL view displays the Code Pattern Model as a part of the cartography tool (step 6 of the "base migration process"). This section presents an approach that identifies PL/SQL code across the legacy and classify it in categories –that appear in the aforementioned view– for developers to decide whether it is worth translating the code or skip it (for example, for obsolescence reasons). In case developers decide towards the translation option, the approach embeds the original PL/SQL in strategical locations of the target code

for programmers to manually migrate the code. The benefit of this proposal is to decrease the effort of identifying and classifying PL/SQL code scattered in different places of the legacy. This section consists of two parts: the first part presents the definition of the code patterns categories, and the second part elaborates on the technical details of the PL/SQL code discovery algorithm which is built on top of the Code Patterns Model.

5.1. Definition of a Code Patterns Catalog

A code pattern is a set of statements that frequently appear together in the code and that implement a specific business logic validation or functionality. For instance, in master/detail forms, the code that prevents orphaned records is common. In this section, we present the categories and patterns of the catalog and describe how such a catalog was built.

5.1.1. Categories and patterns

Patterns are classified into the following four categories: 1) *Field validation*: ensures that the application receives correct and useful data; 2) *Field population*: derives new information from user inputs; 3) *Model constraints*: enforces entity integrity constraints, such as foreign keys and unique keys; and 4) *Miscellaneous*.

This classification was influenced by the strategy followed to build the catalog. Firstly, we manually identified common patterns in a dataset of applications provided by Asesoftware. After that, we relied on experts opinion to improve the patterns list. Finally, for the sake of alignment, we compared our patterns with the classification that the same Oracle Forms community gives to the triggers (e.g., WHEN_VALIDATE; ON_POPULATE and ON_CHECK).

Each pattern is described by means of a template with the following items: abbreviation, name, purpose, pseudocode (i.e., the pattern specification), and parameters (the parts of a pattern that are dynamic). Note that some patterns can be grouped in families of closely related syntactic variations that express the same functionality; e.g., a SELECT-INTO or FETCH statement could be used to retrieve records from the database without changing the code semantics. For illustration purposes, the pattern that describes the code given in Listing 1 is presented below. Remaining patterns are in Table 6.

DEL_VAL, Delete validation in Master-Detail relationship
Purpose: To validate if there is any detail record when the user is trying to delete a master record [3].
Pseudocode:

```
CURSOR_DECLARATION cursor as
  SELECT 'X'
  FROM tableName
  WHERE col1 = fieldA
    AND col2 = fieldB
```

[3]We used a cursor in this pattern on purpose to show a pattern using this construct instead of the direct SQL sentences shown in similar patterns before. All these patterns can be found in both variations.

```
    AND ....

FETCH_RECORD cursor INTO localVar
IF recordFound
  SHOW_MESSAGE(msg)
  RAISE_ERROR
```

Parameters:
- *fieldA, fieldB, ...*: The primary key of a master record.
- *tableName*: Detail table.
- *col1, col2, ...*: Foreign key of detail table.

5.1.2. Building the Pattern Catalog

We carried out the following process in order to build the patterns catalog: ✂ Remark 2

1. Manual review of 72 triggers out of 11880 instances found in three different large real-life applications. These triggers were randomly selected by using the Excel function. We checked that the sample included a number of instances that falls in categories well-known by the Oracle Forms community. It is worth noting that some instances were out of these pre-existing categories and motivated us to include new taxonomies that reflect particular code conventions.

2. Documentation of each trigger, including: form name, GUI object that triggers the event, event type, and description of the intended functionality.

3. Grouping of triggers according to intended functionality.

4. Selection of the candidate groups to become code patterns.

5. Evaluation of candidates by Asesoftware experts.

The remaining paragraphs describe the evaluation in detail since it was the step that allowed us to improve the classification.

We asked four Oracle Forms experts from Asesoftware —two with professional experience that ranges from 5 to 10 years and two with more than 10 years experience, and none of them co-authors of this paper— to validate the correctness of the identified pattern occurrences. We scheduled a four-hour meeting to accomplish this task.

Due to time constraints, and based on an estimated effort of 10 minutes to validate the correctness of a given occurrence, we randomly chose 72 of the pattern occurrences discovered in the studied applications, ensuring that every pattern was present in the sample. Then, we distributed this sample among the four experts. Finally, we prepared the following evaluation instruments: 1) a file that consolidated the PL/SQL code of the selected occurrences; and 2) a spreadsheet that classified each occurrence into a pattern type and described the code pattern model elements created from the PL/SQL code. Each pattern occurrence was reviewed by only one expert, however, each pattern type was reviewed by two or more analysts.

During the review session the experts answered the following questions for each pattern occurrence: 1) Is the PL/SQL code properly matched against the pattern?; 2) Is the data collected in the code pattern model elements sufficient enough to reproduce the functionality in the target technology?; and 3) Do you have any comments or suggestions? Table 1 shows the findings reported by the team of experts at the end of the session, which helped us improve the classification. The table has two columns: the first one reports the findings and the second column classifies the findings in two categories (either wrong matching or improvement suggestion).

Table 1. Findings report

Findings	**Type**
Sentences like "SELECT function INTO:var FROM DUAL" are basic assignments so that the pattern is "Populate a form variable with a functions return value" instead of "Populate a form variable from SELECT statement".	Wrong matching
The "Logging" pattern must take the context into account, because not every "INSERT INTO" sentence aims at recording user or application events.	Wrong matching
The pattern TEX_MSG (text message management) must take the context into account, since a message could be used to disable transactions (e.g., an insert or an update).	Wrong matching
The pattern CMP_VAL (Greater/Less vali-dation) ignored the guard expression in the "ELSIF" statement.	Improvement Suggestion
The pattern RNG_VAL (numerical range validation) ignored the guard expression in the "ELSE" statement.	Improvement Suggestion

5.2. Discovery of PL/SQL Patterns Occurrences

5.2.1. Parsing PL/SQL statements embedded into OF triggers

First of all, we defined a grammar that builds an Abstract Syntax Model (ASM) from PL/SQL code. The grammar conforms to "The Database PL/SQL Language Reference"[27], including variables and constants declarations, data types, control statements, SQL statements, error handling, cursors, blocks, and comments. From this grammar, XText[28], which is a framework for developing Domain-Specific Languages(DSL), was used to generate a parser and a specific PL/SQL ASM metamodel (after some tweaks to optimize the result).

5.2.2. Identifying code patterns

A transformation takes the ASM and converts it into a Code Pattern model containing the information about all matched patterns in the legacy application.

Code Pattern concepts. A simple metamodel to collect the relevant information for each detected code pattern is shown in the upper part of Figure 8) with gray background. The Figure also shows how these concepts are related to the intermediate metamodel, which is introduced in [9]. Note that we do not keep the source code itself but only the relevant pieces of information that, together with the type of pattern identified, suffice to understand the goal of that code excerpt. Main elements of this metamodel are:

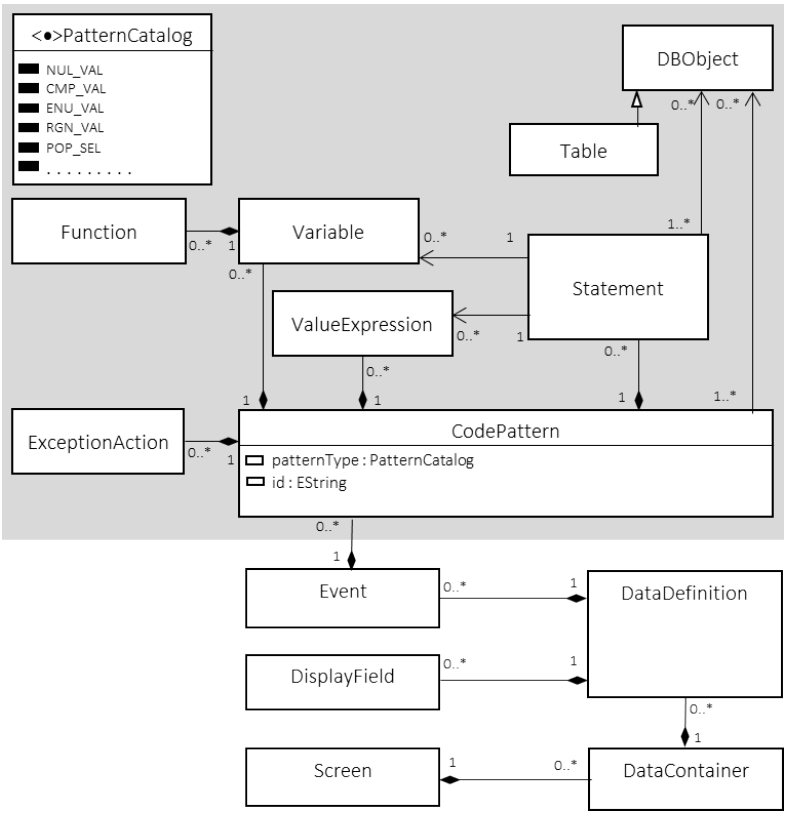


Figure 8. Code Pattern and Intermediate Metamodels.

- *Code Pattern* is the central class ended to access to all relevant data (e.g., the matched elements) and storing the *patternType* attribute. The m2m transformation –that merges the intermediate model and the code pattern model– takes advantage of this concept to relate PL/SQL elements and legacy structural elements (i.e., Events, Fields, Data Definition, Data Containers and Screens).

- *Statement* represents the key instructions of the trigger code; e.g., to invoke database procedures or data manipulation operations, such as aggregation and filtering.

- *DBObject* represents a database object referenced from a Statement.
- *Value Expression* specifies a simple operation that involves arithmetic or logical operators.
- *Variable* represents the value stored by a Display Field which has been retrieved or modified by the trigger code.
- *Function* represents a function applied to a Variable, such as "convert to upper case" or "round".
- *Exception Action* represents the action taken when a specific condition occurs. In most cases, the action breaks out the execution flow and displays a message to the user.

5.2.3. Matching algorithm

In this step, we apply an algorithm that matches the catalogue of patterns against the ASM. The algorithm consists of a set of *matching operations*, where each operation aims to identify instances of a single pattern. This facilitates the addition of new patterns to the discovery process. Each trigger can match zero, one, or many patterns (e.g., the first lines of a complex trigger can match a pattern while additional lines can match a different one) but patterns do not overlap so there is no need to execute the discovery operations in any particular order.

As an example, listing 2 shows the discovery operation *patternDelValFetch* corresponding to the DEL_VAL pattern. Given the ASM of a trigger, the operation enriches the contextual Event if the three following fragments of code are found in the trigger: 1) a fetch statement which retrieves a Forms variable that refers to an item of the user interface (lines 5 to 7); 2) a cursor declaration which is used in the fetch instruction and has a filter on the Forms variable (line 9); and 3) a segment of code that displays an error if the fetch statement has returned a record (line 10). Again, this operation covers one of the syntactic variations of the pattern. When necessary, more than one discovery operation per pattern has been developed. Due to each operation takes as input the ASM of a trigger it is possible to reach any parameter required in the matching operation.

Snippet 2. Discovery function for the DEL_VAL pattern

```
1 operation Event patternDelValFetch (trg: Trigger) {
2  var query : Statement;
3  var pattern : CodePattern;
4  var exception : ExceptionAction;
5  for(fetch in allStmts(trg, FetchStatement).select
6                (f | f.variables.size() = 1
7                     and f.containsFormsVars()))
8  {
9   query = existsSimpleCursor(trg, fetch);
10  exception = existsExceptionAction(trg, fetch);
11  if (query.isDefined() and exception.isDefined()){
12    pattern.statements.add(query);
13    pattern.dbobjects.add(query.getTable());
```

```
14      pattern.exceptions.add(exception);
15      pattern.category = PatternCategory#DEL_VAL;
16      self.patterns.add(pattern);
17    }
18  }
19 }
```

Figure 9 shows the code pattern elements created as a result of matching the trigger element against the *patternDelValFetch* operation. Note that the function classified the trigger as an occurrence of the DEL_VAL pattern.

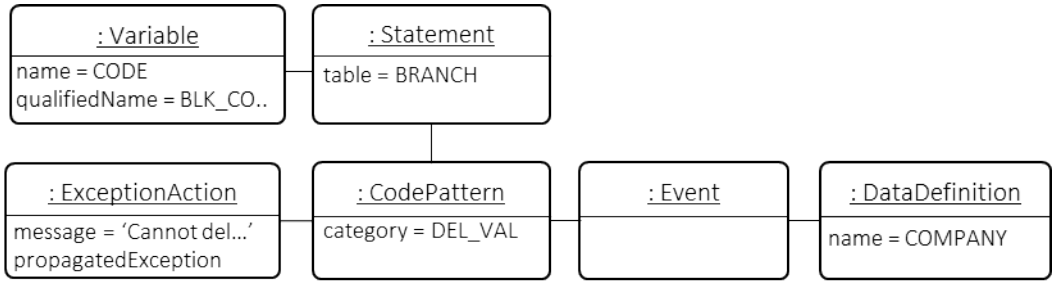


Figure 9. The Code Pattern Model for the trigger example

The algorithm has been implemented as a combination of imperative Java code with declarative model-to-model transformations.

6. Applying the migration process to case study

As aforementioned, the University team worked in collaboration with Asesoftware (the industrial partner) to carry out the "base migration process" (see Fig. 1 a) for OF applications. After that, Asesoftware applied the migration approach and tooling to three Latin American companies in the sectors of banking and insurance, thus instantiating the second process (see Fig.1 b). Asesoftware reached the step of "pre-sale" in two of these companies and fully executed the process in the third. We observed that new migration projects might motivate changes in the tooling produced in the base migration process.

Table 2. Components prone to changes.

Tooling components	**Underwent changes?**
Parsing and post-processing	Yes
M2M Transformation legacy model 2 intermediate model	No
M2T Transformation intermediate model 2 code	Yes
Estimation tool	Yes
Cartography Tool	No

Table 2 shows whether tooling components required adaptations or not. On one hand, the cartography tool and M2M transformation were kept as-is. The proposed views satisfied the comprehension needs. In addition, the intermediate model had all the necessary abstractions to guarantee the code generation as expected; as a result, there was no need for modifying the M2M transformation. On the other hand, we carried out the following developments:

- We developed new parsers to interpret different Oracle Forms versions (i.e., 6i, 9i, 10i, 11i). The variations among them are related to the internal representation of items; therefore, we had to guarantee that items (regardless of the version) were homogeneously represented in the legacy model in order to avoid negative impact in the M2M transformation.

- We implemented post-processing routines that enriched the legacy model with information coming from additional sources (e.g., database scheme). The reason was that certain data access information —needed in the modern application— was missing at the application level and, thus, at the legacy model. For example, primary keys can be optional in Oracle Forms (instead of using unique indexes) but mandatory in JPA. In that case, our post-processing routine extracted this information from the scheme and injected it in the legacy model.

- We developed new M2T transformations to generate code in the technology required by clients. Clients mainly highlighted the need for supporting diverse UX technologies (i.e., Java Primefaces, .Net).

From this experience, we concluded that parsers, post-processing routines, and M2T transformations are the components most prone to adaptations when a new project comes up. For the sake of maintainability, the MDE expert should provide with extension points to facilitate the connection and switching of these components.

In the remaining parts of this section, we focus on the process application because it shows the benefits of the tooling built in the base migration process. To this end, we chose a client where the process was fully instantiated: a Bank that needed to migrate distraint software. We took advantage of this project to: i) have a case of success that serves as a reference when commercializing the service; in this case Asesoftware negotiated charging the Bank for only a small percentage of the actual migration cost in exchange for using the project as a business case; ii) shed some light on developer productivity when using our process: Asesoftware needs this information to define how to monetize the modernization service; and iii) evaluate tooling functionality.

6.1. General description of the Bank project

We applied the approach in the Bank project where the purpose was the migration of a *Seized Asset Management System* comprised of 32 forms, 116 blocks, 1929 items and 1647 triggers.

This project was a win-win for both parts: the Bank and Asesoftware. The Bank was so satisfied with the process and the product that it launched a bidding process to hire services

for modernizing other legacy systems. In turn, Asesoftware built its first case of success from this experience and opened doors to new contracts with the Bank.

Asesoftware formed a team of seven people to carry out the project: a software architect, an Oracle Forms senior developer, an MDE expert, and four system analysts that had experience in the target technology but were not Oracle Forms experts. The first three people already knew the relevant technologies and tools. In contrast, the analysts were novices in the context of migration. Therefore, they received a training in Oracle Forms, in the target architecture, and in our migration process.

6.2. Results and Discussion

This section is divided in two parts. First, we highlight the savings obtained in each step of the migration application process for the Bank project. After that, we present the economy resulting from the PL/SQL pattern matching algorithm and its associated view. We discuss the results at the end of each part for the sake of concluding one aspect at a time.

6.2.1. Productivity in the migration application process

The team invested 4245 hours in the migration of the legacy. Table 3 shows time savings in each step of the process. An overall comment from the development team was that the tooling helped the collaborators to increase their productivity and smoothed their work when compared with the manual previous process.

Table 3. Time savings for each process step.

Process Step	**Time Savings**
Pre-sale	80%
Analysis	42.65%
Design	78.8%
Construction	16.5%
Testing	43.45%
Overall	**31%**

The steps with the highest time savings were "pre-sale" and "design" (80% and 78,8%, respectively). In "pre-sale", the effort decreased from 40 to 8 hours. This time is devoted to executing the cartography and estimation tools and building an executive presentation including the results. In "design", the gains were high because most of the architecture decisions where made in the base migration process. As mentioned in section 2.2, the decisions made at this step aim at re-architecting the application by using the configuration features provided by the development team.

In "analysis", time savings were of 42,65%. The main reason was that the MDE expert needed to develop and plug new operations to the matching algorithm in order to produce a Code Pattern Model as purged as possible.

In "testing", time savings were of 43,45% because the current tooling generates only unit tests for the business layer. Instead, the team manually performed functional tests.

The step with the least time savings was "construction" (16,5%) this is due to: i) the application had 3 forms that did not fall in our initial typology of programs susceptible to migration (i.e., forms with either a master table or two tables associated via a master/detail relationship); instead, they were complex forms, each of them, with multiple tables. The team had to manually migrate these forms; and ii) the client required a UI design that was different from the default one; as a consequence, new model-to-text transformations were developed to generate Web pages satisfying such a design.

Table 4 compares the effort, cost, and schedule of the manual method with those of the SMoT method. In addition, the table shows the savings in these three dimensions. It is clear that some values (e.g., those of the "construction" step) introduce noise to the savings. However, these results gave Asesoftware an initial idea about the margin of discount (applicable over the estimation effort) that they may offer to potential clients. This saving complemented by all the features of our white-box modernization approach may give a competitive advantage over companies offering similar services. Asesoftware plans to adjust saving rates as more projects are carried out.

The effort to build the tooling was around 5000 hrs. We excluded the hours of MDE training in this count because the development team was enrolled in a master course about MDE before the "Base migration process" got started. Therefore, if we add to 5000 hrs the effort invested in the "instantiation process (i.e., 4245 hours), we arrive to a total effort that almost doubles that of the manual migration (i.e., 5126). The "Base migration process" requires an upfront investment because it is necessary to develop the MDE infrastructure required in the re-engineering, restructuring, and forward stages. But this will be beneficial for the company because:

- The quality of the new code is significantly higher when following the white-box transformation than when applying the manual migration (environ 61 %) [9].

- The cost of migrating by hand is mainly influenced by the size of the legacy. On the MDE alternative, the size of the legacy only impacts some steps in/of the "instantiation process". On the one hand, the cartography, estimation, and actual translation from legacy to target are not impacted by the size of the legacy, e.g., in our case study, it took less than one hour because the output was directly obtained from the legacy and intermediate models, without human intervention. On the other hand, the time devoted to the steps of analysis, design, code completion, and testing depends on the number of forms and their complexity. The higher the number of forms and their complexity, the higher the time that developers have to devote to navigate the views, make decisions on top of them, or implement the remaining code (i.e., the one not generated automatically by the tool, e.g., PL/SQL code). However, this time is less compared with that of the manual approach. As an example, we experimented with a low complexity (i.e., less than then UI items) and a high complexity (i.e., more than 20 UI items) forms. The times required to migrate the forms are: 31.4 and 82.4 hours,

respectively, with the manual migration, and 18.1 hrs and 41.5, respectively, using the white-box approach.

- Asesoftware's team has carried out the steps of cartography and estimation in three different projects and reported a significant reduction on effort investment. Here, we evidenced that MDE techniques pay-off the upfront cost when gradually used along different projects ([29]).

Table 4. SMoT migration savings over manual migration.

	Effort (Hours)	**Costs (K $)**	**Schedule (Months)**
Manual Migration	5126.2	154	7.7
SMoT Migration	4245	106	5.3
Savings	881.2	48	2.3

6.2.2. PL/SQL view in practice

Table 5. Number of triggers marked as "to skip"/"to migrate" in iterations of real project

	First *	**Second** **	**Third** ***
Numbers of triggers marked as "to skip"		929	223
Numbers of triggers marked as "to migrate"		718	495
Number of triggers under evaluation	1647	1647	718

* Fully automated
** Semi-automated
*** Manual

This experience demonstrated that the analysis of the results of the PL/SQL view is an iterative process, since there is no a definitive catalog of patterns applicable to all applications. Instead, the MDE expert likely needs to develop and plug new operations to the matching algorithm in order to produce a Code Pattern Model as purged as possible. The gist is to increase the productivity of Oracle Forms experts when they mark the triggers as "to skip" or "to migrate".

Table 5 summarizes the project results. Note that there were three iterations.

In the first one, the default matching algorithm (the one configured to match the patterns of our catalog) automatically computed a Code Pattern Model containing 1647 occurrences of triggers. The Oracle Forms expert randomly selected some triggers from the model in order to review and identify patterns that reflected the company's coding style. The number of selected triggers were constrained by the hours initially estimated for this task; that is, eight

hours. She observed triggers that met two behaviours: i) implement standard functionality (for example, CRUD, opening of a form, etc.); or ii) are launched through UI facilities that are out-dated. For example, basic functionalities like saving or navigating a record back/forward are triggered via the keyboard. In our migration step, these functionalities are generated by default as part of the CRUD and programmed to be launched through the mouse, instead of keyboard shortcuts. The Bank agreed this feature; as a consequence, it did not make sense to migrate these types of triggers.

Based on the legacy expert output, the MDE expert developed matching operations to identify occurrences of the triggers mentioned in the previous paragraph. These operations were plugged to the default matching algorithm resulting in a new one referred to as "Matching algorithm V2". In the second iteration, the legacy expert executed the "Matching algorithm V2" and got a Code Patterns Model where 929 triggers were marked as "to skip" and 718 marked as "to migrate". Then, instead of having to check 1647 triggers (i.e., the total number of triggers found in the legacy) to decide which are worth migrating, the expert only checked 718, which corresponds to 46% of the total triggers. This represents time savings in this task.

We say this iteration was semi-automated because the marks automatically computed by the matching algorithm depended on a previous manual review made by the legacy expert. In the third phase, the legacy expert focused on reviewing only the collection of triggers marked as "to migrate" in the second iteration. The purpose of this was to further evaluate if such a collection contained triggers that could be skipped in the migration step. As a result, she marked 223 triggers as "to skip" and 495 marked as "to migrate".

The matching algorithm matched 87% of the 495 triggers "to migrate" with categories of our catalog. For the remaining 13%, the algorithm did not find correspondences between the triggers and the existing patterns. Figure 10 shows the variety of patterns found in the 87% of the triggers. 39.2% of occurrences were classified in the Oracle Forms proprietary routines (i.e., FRM_PRP). Along the iterations, the legacy expert marked FRM_PRP triggers as "to skip/to migrate". This shows that it is not necessarily good to mark as "to skip" by default the triggers that match a given pattern. Instead, it is desirable to have a review from the expert before making a final decision.

The migration step embedded these 495 triggers as commented code in assets of the target application. Pattern classification was also useful for the development of tests.

When comparing triggers skipped in the last iteration with those of the first iteration, we found that all of them encapsulated similar behaviour. The differences were the Oracle Forms proprietary routines used in the implementation. Maybe if the legacy expert had had more time in the first iteration, then she could have discovered more patterns early, and then could have had a more decanted collection in the last iteration that would have sped up the review.

The MDE expert developed matching operations for the patterns discovered in the last iteration. All matching operations were saved in a repository ready to be used in the migration of another application from the same client.

Type	#Occurr.	%
CLR_FRM	0	0.0%
USR_OPT	0	0.0%
SYN_DET	0	0.0%
POP_SEQ	0	0.0%
DEL_VAL	0	0.0%
LOGG	0	0.0%
UNQ_VAL	1	0.2%
POP_INP	2	0.5%
DOM_VAL	3	0.7%
POP_DFL	4	0.9%
CPX_VAL	4	0.9%
EXT_VAL	7	1.6%
CMP_VAL	8	1.8%
POP_FUN	10	2.3%
TEX_MNG	18	4.1%
RNG_VAL	26	6.0%
POP_SEL	56	12.9%
PRC_INV	59	13.6%
NUL_VAL	66	15.2%
FRM_PRP	170	39.2%
	434	**100.0%**

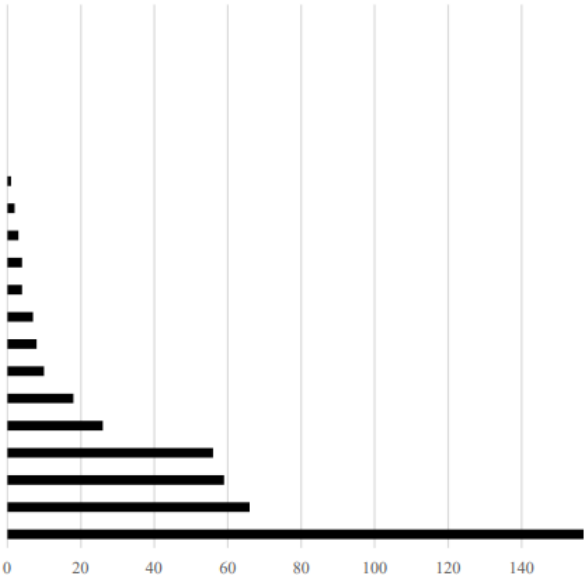


Figure 10. Pattern occurrences classified by type for triggers marked as "to migrate" in the second iteration

7. Lessons Learned

In this research, we have identified several lessons learned concerning the use of MDE techniques, code patterns, and academia-industry collaboration:

7.1. Use of MDE techniques

- In our approach, the PIMs support users on: i) architecture decision making even in cases where users are not legacy experts; and ii) sanitization of the legacy model information. In addition, the PIMs eased plugging new parsers and model-to-text (M2T) transformations. The parsers provide a representation of legacy applications, but in terms of the PIMs concepts. We developed parsers for different OF versions. In turn, the M2T transformations translate the PIMs into an application written in a particular target technology. We implemented M2T transformations towards Java and .Net. Therefore, no effort should be spared to design a robust PIM for the family of technologies targeted in the migration. To this end, we suggest to review variations on the source and target technologies.

- The (partial/full) application of the migration tooling to three Latin American companies showed that said tooling –produced in the base migration process– can be mostly reused in many migration projects within different company's clients. In particular, PIMs, their associated graphical editors and model-to-model transformations can be reused as is. Furthermore, the application of the approach to the Bank project also demonstrated that clients may have particular inquiries concerning the source or target technologies that may impact the tooling. For example: i) variations among source

technology versions (e.g., OF versions: 6i, 9i, 10i, 11i) may imply slight changes in the parsers; or ii) modifications in the target architecture –to adjust the applications look-and-feel or security mechanism– may involve changes in the model-to-text transformations. Here, it is important to have a robust configuration management of the tooling.

- UI is a key point to validate because most clients want to impact user experience as little as possible. In the pre-sale step, the team shows the demo to the client and asks her whether the target UI meets the requirements. This helps the MDE expert to early realize if she needs to modify an existing bridge (or build a new one) from the PIM to the target. If yes, the project manager should modify the project budget too. MDRE projects should include agile practices that allow the migration team to have early feedback from the client and adjust the tooling accordingly.

- In this case, the transfer of MDE techniques to industry was a reality. Indeed, Asesoftware created an automation department where developers apply MDE techniques on their own to solve company's challenges. Besides the aforementioned items, we think that the definition of early victories in the base migration process plan helped Asesoftware to gain confidence in MDE based proposals.

- There are plenty of factors that influence the success or failure of adopting an MDE approach in a company [30]. In this case, the project was a success and we attribute this success to: i) a formal training to the company employees by university experts on MDE; ii) involving management members as project stakeholders; and iii) a clear scope and a shared vision of the project. First and second factors are in accord with Hutchinson et al. [31], where the authors point out that the success or failure of MDE is significantly affected by factors such as training and commitment to the project.

- Despite the positive view of MDE, we did run into some troubles when trying to use existing MDE tools in an industrial setting. As an example, even if declarative transformation rules as the ones provided by ETL are conceptually better and simpler, we found that they were not expressive enough to cover all our needs and therefore we were forced to "pollute" ETL code with imperative Java constructs to manipulate strings and fine-tune the rule application ordering. Another example would be the building of the XText-based abstract syntax metamodel for PL/SQL. There, we had to go back and forth between the grammar and Xtext in order to solve ambiguity problems and get a readable metamodel. Finally, we developed unit tests for the m2m transformations. Such tests aim at detecting regressions by comparing gold standard models with the models outputted by the last versions of the transformations. In this part, we encountered that Epsilon Ant tasks have problems when loading models produced by XText. We arrived to a workaround that programmatically calls the tests. This last point led us to one reflection: practitioners perceive that the maintenance and documentation of MDE tools is poor, which may discourage them from using the

MDE paradigm. This thought is also mentioned in [30]. It is key to involve MDE experts in the projects to alleviate the lack of documentation and maintenance efforts.

7.2. Code patterns in industry

- Our catalogue should be taken as a starting point when facing a new migration project but not as a closed set. For instance, Asesoftware opted for using extensively stored procedures as a way to isolate and reuse business logic across forms. As we discussed before, this code was deemed unnecessary to transform but for companies that would like to migrate everything, patterns should be extended to adapt to such company policies (e.g., coding style and conventions) and cover all the aspects they need to migrate.

- Our catalog has patterns that express general good practices when developing RAD applications. However, from our experience in real projects, we learned that every software house may define their our patterns and implement the code accordingly. It is up to MDE experts to evaluate the cost/benefit of modifying the discoverer to deal with customized patterns. Experts make the decision taking into account, for example, the size of the applications. Therefore, PL/SQL pattern discoverer has to be flexible enough to allow MDE experts to add new patterns in it.

- We defined a very narrow migration scope (only master and master/details forms) that prevented us from (semi)automating the transformation of grid-style forms, and forms with multiple details, which are actually quite popular in OF applications. In the step referred to as "definition of migration strategy" in the base migration process, it is important to analyze existing repositories of legacy code to identify the most commonly used constructions and their correspondence in the target technology. This helps the development team to choose which legacy constructions deserve translation to the target.

8. Related Work

Legacy migration often comprise reverse and forward engineering techniques. The study of these techniques dated back many years ago. For example, Selfridge et al. [32] spelt out reverse engineering challenges in early nineties. Several approaches have appeared from this date up until now. The related work found for this paper can be divided in three main categories:

- **Legacy-to-target transformation:** works under this category mainly aims at moving systems from one language environment and platform to another. We refer to this task as the "transform" step in our *migration application process*.

- **Migration process management:** includes works that center around the migration process in overall.In the literature, most works focus on the technical aspects of the legacy-to-target transformation. However, in this category, we are interested in the

works that tackle a wider perspective that covers not only the transformation but also the management of a migration project.

- **PL/SQL reverse engineering:** covers works that discuss the reverse engineering of PL/SQL code and migration to modern technologies by using MDE.

Even tough the focus of this paper goes beyond the legacy-to-target transformation, we present the works classified in the first category in a summarized form, for the sake of noting what has been done in this respect. After that, we detail the approaches that fall in the second and third categories which are actually closer to our approach. Each following section offers a comparison between the presented works and our approach.

8.1. Legacy-to-target transformation

We found academic approaches that focus on the migration of *Scientific* and *Transactional* applications. Scientific Applications are mainly developed to better understand or make predictions from data that describes real world processes. In turn, Transactional Applications are developed to ensure that an organizations' data is recorded in the database in accordance with the corresponding transactions.

To ease the comparison of our approach and these approaches, we use the following criteria: i) the source and target technologies of the migration; and ii) the human intervention during the migration process to determine if the reviewed approach falls into our definition of White-Box or Black-Box.

Approaches tackling Scientific applications [33, 34, 35] migrate programs written in languages such as C/C++ and FORTRAN-77 to Java and object-oriented designs. In turn, approaches tackling Transactional applications [36, 37, 38, 39, 40, 41, 42, 16, 43, 44, 45, 46, 47, 48, 49, 50] migrate from mainframe technologies (e.g., IBM Assembler, COBOL) and Fourth Generation Languages (e.g., Oracle Forms) to Web technologies, in some cases to be complaint with SOA or cloud architecture patterns. Even tough, our case study fixed the translation from Oracle Forms to Java and .Net, the approach has been designed in an agnostic way for the sake of being applied to other source/target technologies.

Nearly all the reviewed approaches [36, 37, 38, 39, 40, 41, 42, 16, 43, 45, 46, 49, 47, 48, 50] use some kind of intermediate representation to decouple the input and output. In most cases (except by [43]) the target architecture is unique. Our approach also includes this feature of intermediate representation and, even if the architectural patterns are pre-defined, it is possible to configure some aspects —e.g., maintainability, usability, and security— during the early stages of the migration cycle.

From a commercial perspective, we limit the comparison to the tools that transform Oracle Forms applications. The transformation is usually offered as a service such as in QAFE[1] or as a standalone platform ready used by the developer (i.e., Pitss[2], Kuraman[3], Evo[4]). Some of the tools like Evo and Jheadstart[5] transform Oracle Forms to an in-house technology such as ADF or APEX (i.e., EVO and Jheadstart.). Others like Composer, Ormit[6], Composer[7], Jheadstart, Composer, Ormit, Evo and Turbo[8] have a completely different target technology such as .NET or Java.

In brief, we classify the reviewed approaches (except [45, 43]) into the black-box transformation category because they generate applications that conform to a default architecture instead of letting the user decide about some aspects. As a result, users can only specify their decisions over the generated code. In contrast, we provide the means to make decisions in the early steps of the transformation by using a model and an editor. Similarly, [45, 43] have a model that represents the application in an agnostic way; however, the papers do not report any high-level abstraction means to make decisions based on such a model and follow the migration project process, which our approach does make possible via our graphical editors.

8.2. Migration process management

Fleurey et al. [4] define a model-driven (MD) migration process, which is divided into four steps: Parsing, Reverse engineering, Transformation, and Code generation. This process uses three (meta)models: legacy model, ANT model (kind-of PIM), and PSM model. The legacy model represents the aged technology. The ANT model represents concerns such as: graphical interface, application navigation and data structures. The PSM models the target platform. The legacy model is converted to the ANT model, which is in turn translated to the PSM model. The authors also implement a modeling platform adaptable to different legacy and target technologies and TODO directives to indicate developers which parts require manual intervention. This migration process is part of a larger migration perspective that includes four phases. The first phase is a technical analysis of the project that includes a sample migration, cost estimation, and first client proposal. The second phase is the development of model transformations, parsers and generators. The third phase is the pilot project that includes another sample migration of a representative part of the legacy application, and produces the final cost of the migration project. And finally, the last phase is the actual migration process of particular applications.

Fuentes et al. [51] also propose a migration process. They define a metamodel (referred to as XSM) representing the different levels of abstraction of a modernization process. This process includes four phases. The first phase is the Preliminary evaluation of the project, where experts: i) collect relevant information about legacy and target platforms; ii) estimate the costs of the project; and iii) propose a modernization offer to the client. In the second phase (referred to as Understanding), the identification of application requirements and core components is made, and the XSM models of the legacy application and target platform are created. The third and fourth phases are Building and Migration, which refer to the implementation of transformations and the execution of the transformations to complete the migration process. They propose a feature-driven approach for the modernization, applying the migration process to specific features in an iterative way.

Fuhr et al. [52] present an approach for the migration of legacy systems to SOA (Service-Oriented Architecture) by using an extension of the IBM SOMA (Service-Oriented Modeling and Architecture) method. In the extension, the authors use TGraphs to represent the artifacts and graph-based queries and transformations for the re-engineering and conversion processes. The TGraph representation of the project is analyzed in conjunction with activity

diagrams models of business processes to identify services in the legacy code, and then generate corresponding services in a SOA platform.

Wagner et al. [53] present a model-driven approach for re-engineering and migration of software projects. The re-engineering step consists of the parsing and conversion of the source code into an annotated AST. Annotations add extra information about data flow and types. Furthermore, this step creates a platform-independent description of the source system, which serves as an input for the automatic creation of models that reflect different perspectives of the legacy. The final stage of the re-engineering process is called "model analysis" and its purpose is to generate graphical illustrations that help developers to be aware of static metrics, dependencies between functions, and dead code. In the migration step, developers manually derive a "process model" from the models obtained in the re-engineering step. Finally, a code generator produces the target application from the process model.

Bermudez et al. [54] contribute a tool called Models4Migration that partially automates the migration process of software projects. This tool transforms a given legacy into an "Abstraction model". This model conforms to a metamodel that is reused in each migration project, independently of the source technology. Then, the tool transforms the "Abstraction model" into a "Concrete model" that represents the legacy but in terms of the target technology concepts. The tool receives as input the (meta)models, the transformations, and a specification of the transformation chain. The chain is specified through a DSL defined by the authors. The tool can reuse the inputs to migrate different projects that have the same source and target technologies. The approach encompasses a stage where developers have to manually modify the target software. To smooth this task, the tool creates requests in a ticket system and assigns developers to each request.

Fleurey et al. [4] only describe the step-by-step construction of a migration tool (i.e., base migration process), while our approximation also describes how to use the migration tool in projects (i.e., instantiation process). In [4], the cost estimation is manually done, while our approach (semi)automates effort calculation. We share some similarities, such as highlighting the parts in the generated code that need manual inputs from developers and the implementation of adaptable modules to tackle different source/target technologies. Fuentes et al. [51] propose an iterative approach at the base migration process, while our approach is only iterative at the instantiation process. This is actually a "nice to have" feature that we would extrapolate to our approach. Fuhr et al. [52] center around a single type of target architecture, while using graph-based transformations. Bermudez et al. [54] and Wagner et al. [53] describe generic approaches to software migration, with the inclusion of the instantiation process for each project. Bermudez et al. [54] goes a step further [53] because it adopts the use of ticket systems to manage the tasks that developers have to manually perform in the target application.

8.3. PLSQL

Regarding PL/SQL coding in a broad sense, [14, 15, 16] reverse engineer large legacy PL/SQL applications to achieve program comprehension and refactoring but do not address the migration of such code.

As far as we know, only [17, 18, 19] deal with the reverse and forward engineering of PL/SQL code embedded in Oracle Forms applications. They focus on modeling PL/SQL language structures at a syntactic level.

Like [14, 15, 16], our scope is to reverse engineer the PL/SQL code with a program comprehension intend. In particular, SMoT classifies triggers (according to functional purpose) in order to help analysts decide if the migration of PL/SQL triggers is worth it. The difference with respect to similar works is that our approach goes beyond the syntactic level: it specifically targets the semantics of this domain by first grouping PL/SQL statements that are typically used together. Furthermore, we aim to discover business-level functionality regardless of the number and complexity of statements required to perform it.

9. Conclusions and future work

In this paper, we presented a (semi)automated process for migrating Oracle Forms applications. The process goes beyond the reverse and forward engineering tasks as it encompasses management tasks. The process is supported by a model-based tool called SMoT, which has been successfully transferred to a software company.

In addition to the SMoT features mentioned in our previous work (e.g., structural views, (semi)automated code translation, project tracking) [7, 8, 9], we have added planning project features to the approach. The project manager and client are able to straightforwardly decide the scope and priorities of migration, with the help of specific views that help understanding the structural elements of the legacy architecture (e.g., modules, forms, database tables, relationships, etc.). Furthermore, there is an additional view that collects and classifies the behavioural elements of the application (i.e., PL/SQL routines), helping the legacy expert to decide whether to include a given functionality in the new target system. The idea is to preserve what adds value to the business and peel off those aspects too dependant on the source technology and that are irrelevant in the one.

The case study shows the benefits of our migration process in terms of effort savings and its flexibility to embrace other source and target technologies.

In particular, this study shows time savings that range from 80% to 16% in the different steps of the migration application process, 30% in average. The "construction" was the step with the least time saving and the "pre-sale" and "design" steps had the highest savings. In addition, the study showed that the quality of the new code is significantly higher when using SMoT than when using the manual migration approach (environ 61%).

With respect to the reusability of the tooling produced in the "base migration process", we evidenced that platform independent models (PIMs), their associated graphical editors and model-to-model transformations can be reused as is among different projects. In turn, parsers and model-to-text transformations may require adjustments to tackle variations in the source/target technologies. In particular, for the case study, we had to tune the parser to support different source Oracle Forms versions (e.g., 6i, 9i, 10i, 11i). In addition, we developed diverse model-to-text transformations to translate the PIMs to two target technologies, i.e., Java and .Net.

We also concluded that the proposed PL/SQL patterns catalogue should be taken as an initially point that has to be extended to take into consideration software company policies (e.g., code style, conventions). Indeed, new patterns will require extensions of the matching algorithm.

As future work, we plan to: i) unfold procedure invocations of database code and explore patterns in those stored procedures, since most of that code could also match patterns in our catalogue; ii) implement a code cloning algorithm that could suggest additional refactoring opportunities by signaling repetitive coding excerpts even if not matching any specific pattern in our catalogue; iii) investigate the conversion of PL/SQL code to target technology. If we find a solution that balances benefit and cost, we may explore how to increase the flexibility of the conversion by allowing developers to select among several implementation patterns according to their preferences/context (e.g., data quality, security constraints, internal standards);

References

[1] E. J. Chikofsky, J. H. Cross, Reverse engineering and design recovery: a taxonomy, IEEE Software 7 (1) (1990) 13–17. doi:10.1109/52.43044.

[2] R. C. Waters, Program translation via abstraction and reimplementation, IEEE Transactions on Software Engineering 14 (8) (1988) 1207–1228.

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

[4] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, J.-M. Jézéquel, Model-Driven Engineering for Software Migration in a Large Industrial Context, in: Model Driven Engineering Languages and Systems SE - Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2007, pp. 482–497. doi:10.1007/978-3-540-75209-7_33.

[5] L. F. Mendivelso, K. Garcés, R. Casallas, Vistas arquitectónicas independientes de tecnología para compresión de software, in: XX Congreso Iberoamericano de Ingeniería de Software, Vol. XX, Buenos Aires, Argentina., 2017, p. 1.
URL http://cibse2017.inf.ufes.br

[6] L. F. Mendivelso, K. Garcés, R. Casallas, Metric-centered and technology-independent architectural views for software comprehension, Journal of Software Engineering Research and Development 6 (1) (2018) 16. doi:10.1186/s40411-018-0060-6.
URL https://doi.org/10.1186/s40411-018-0060-6

[7] K. Garcés, E. Sandoval, R. Casallas, C. Alvarez, A. Salamanca, S. Pinto, F. Melo, Aiming Towards Modernization: Visualization to Assist Structural Understanding of Oracle Forms Applications, in: ICSEA 2015, Tenth International Conference on Software Engineering Advances, IARIA, 2015, pp. 86–95.

[8] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Melo, J. M. Soto, White-Box Modernization of Legacy Applications, Springer International Publishing, Cham, 2016, Ch. Model and Data Engineering: 6th International Conference, MEDI 2016, Almería, Spain, September 21-23, 2016, Proceedings, pp. 274–287. doi:10.1007/978-3-319-45547-1_22.
URL http://dx.doi.org/10.1007/978-3-319-45547-1_22

[9] K. Garcés, R. Casallas, C. Álvarez, E. Sandoval, A. Salamanca, F. Viera, F. Melo, J. M. Soto, White-box modernization of legacy applications: The oracle forms case study, Computer Standards & Interfaces 57 (2018) 110–122.
URL https://doi.org/10.1016/j.csi.2017.10.004

[10] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, R. Casallas, Towards the

understanding and evolution of monolithic applications as microservices, in: 2016 XLII Latin American Computing Conference (CLEI), 2016, pp. 1–11. doi:10.1109/CLEI.2016.7833410.

[11] J. García, K. Garcés, Improving understanding of dynamically typed software developed by agile practitioners, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, 2017, pp. 908–913. doi:10.1145/3106237.3117772.
URL http://doi.acm.org/10.1145/3106237.3117772

[12] M. Riley, Choosing the right tool, Oracle Magazine, July-August 2009.

[13] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Synthesis Lectures on Software Engineering 1 (1) (2012) 1–182. doi:doi:10.2200/S00441ED1V01Y201208SWE001.

[14] M. Altınışık, H. Sözer, Automated procedure clustering for reverse engineering PL/SQL programs, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, ACM, New York, NY, USA, 2016, pp. 1440–1445. doi:10.1145/2851613.2851781.
URL http://doi.acm.org/10.1145/2851613.2851781

[15] C. Nagy, R. Ferenc, T. Bakota, A true story of refactoring a large oracle PL/SQL banking system, in: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011), 2011, p. 1.

[16] C. Nagy, L. Vidács, R. Ferenc, T. Gyimóthy, F. Kocsis, I. Kovács, Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system, in: 2011 15th European Conference on Software Maintenance and Reengineering, 2011, pp. 343–346. doi:10.1109/CSMR.2011.66.

[17] L. Andrade, J. Gouveia, M. Antunes, M. El-Ramly, G. Koutsoukos, Forms2net – migrating oracle forms to microsoft .net, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 261–277. doi:10.1007/11877028_8.
URL http://dx.doi.org/10.1007/11877028_8

[18] Ó. S. Ramon, J. S. Cuadrado, J. G. Molina, Reverse engineering of event handlers of RAD-based applications, in: 2011 18th Working Conference on Reverse Engineering, IEEE, 2011, pp. 293–302. doi:10.1109/WCRE.2011.43.

[19] Ó. Sánchez Ramón, J. Sánchez Cuadrado, J. García Molina, Model-driven reverse engineering of legacy graphical user interfaces, Automated Software Engineering 21 (2) (2014) 147–186. doi:10.1007/s10515-013-0130-2.
URL http://dx.doi.org/10.1007/s10515-013-0130-2

[20] R. C. Seacord, D. Plakosh, G. A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[21] M. Jorgensen, M. Shepperd, A systematic review of software development cost estimation studies, IEEE Trans. Softw. Eng. 33 (1) (2007) 33–53. doi:10.1109/TSE.2007.3.
URL https://doi.org/10.1109/TSE.2007.3

[22] G. Ramalingam, 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.

[23] M. Alalfi, J. Cordy, 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.

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

[25] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, P. Charland, A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams, J. Softw. Maint. Evol. 20 (4) (2008) 291–315. doi:10.1002/smr.v20:4.
URL http://dx.doi.org/10.1002/smr.v20:4

[26] T. Richner, 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.

[27] Database PL/SQL language reference (Jun. 2016).
URL https://docs.oracle.com/cloud/latest/db112/LNPLS/toc.htm

[28] Xtext - a programming language framework (Jun. 2016).
URL http://xtext.itemis.com/

[29] O. Diaz, F. M. Villoria, Generating blogs out of product catalogues: An mde approach, Journal of Systems and Software 83 (10) (2010) 1970 – 1982. doi:https://doi.org/10.1016/j.jss.2010.05.075.
URL http://www.sciencedirect.com/science/article/pii/S0164121210001469

[30] A. Vallecillo, On the industrial adoption of model driven engineering - is your company ready for MDE?, International Journal of Information Systems and Software Engineering for Big Companies 1 (1) (2014) 52–68.

[31] J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, Empirical assessment of MDE in industry, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 471–480. doi:10.1145/1985793.1985858.
URL http://doi.acm.org/10.1145/1985793.1985858

[32] P. G. Selfridge, R. C. Waters, E. J. Chikofsky, Challenges to the field of reverse engineering, in: [1993] Proceedings Working Conference on Reverse Engineering, 1993, pp. 144–150.

[33] Achee, B.L. and Carver, Doris L, Creating object-oriented designs from legacy FORTRAN code, Journal of Systems and Software 39 (2) (1997) 179–194.

[34] M. Bysiek, A. Drozd, S. Matsuoka, Migrating legacy fortran to python while retaining fortran-level performance through transpilation and type hints, in: 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), 2016, pp. 9–18. doi:10.1109/PyHPC.2016.006.

[35] L. Favre, C. Pereria, L. Martinez, C. Pereira, Modernizing Software in Science and Engineering: From C/C++ Applications to Mobile Platforms, in: ECCOMAS Congress 2016 VII European Congress on Computational Methods in Applied Sciences and Engineering M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris (eds.) Crete Island, Greece., 2016, pp. 8162–8176. doi:10.7712/100016.2402.4906.

[36] Z. Tu, G. Zacharewicz, D. Chen, Building a high-level architecture federated interoperable framework from legacy information systems, International Journal of Computer Integrated Manufacturing 27 (4) (2014) 313–332.

[37] Martin Ward, Hussein Zedan, Matthias Ladkau and Stefan Natelberg, Conditioned semantic slicing for abstraction; industrial experiment, Software: Practice and Experience 38 (1) (2008) 1273–1304.

[38] D. Fujiwara, N. Ishiura, R. Sakai, R. Aoki, T. Ogawara, Reverse engineering from mainframe assembly to c codes in legacy migration, in: 2016 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), 2016, pp. 1058–1063. doi:10.1109/IIAI-AAI.2016.37.

[39] Andrea De Lucia, Rita Francese, Giuseppe Scanniello and Genoveffa Tortor, Developing legacy system migration methods and tools for technology transfer, Software: Practice and Experience 38 (13) (2008) 1333–1364.

[40] T. C. Lau, J. Lu, J. Mylopoulos, K. Kontogiannis, The Migration of Multi-tier E-commerce Applications to an Enterprise Java Environment, Information Systems Frontiers 5 (2) (2003) 149–160.

[41] R. Pérez-Castillo, I. G. R. De Guzmán, M. Piattini, Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems, Computer Standards and InterfacesarXiv:1103.2566, doi:10.1016/j.csi.2011.02.007.

[42] A. Navarro, A. da Silva, A metamodel-based definition of a conversion mechanism between soap and restful web services, Computer Standards & Interfaces 48 (2016) 49 – 70, special Issue on Information System in Distributed Environment.

[43] A. Bergmayr, H. Brunelière, J. L. C. Izquierdo, J. Gorroñogoitia, G. Kousiouris, D. Kyriazis, P. Langer, A. Menychtas, L. Orue-Echevarria, C. Pezuela, M. Wimmer, Migrating legacy software to the cloud with ARTIST, in: 2013 17th European Conference on Software Maintenance and Reengineering, 2013,

pp. 465–468. doi:10.1109/CSMR.2013.73.

[44] L. Andrade, J. Gouveia, M. Antunes, M. El-Ramly, G. Koutsoukos, Forms2Net – migrating oracle forms to microsoft .NET, Generative and Transformational Techniques in Software Engineering 4143 (2006) 261–277.

[45] O. Sanchez Ramon, J. Sanchez Cuadrado, J. Garcia Molina, Model-driven reverse engineering of legacy graphical user interfaces, Automated Software Engineering 21 (2) (2014) 147–186.

[46] R. Heckel, R. Correia, C. Matos, M. El-Ramly, G. Koutsoukos, L. Andrade, Architectural transformations: From legacy to three-tier and services, in: Software Evolution, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 139–170. doi:10.1007/978-3-540-76440-3_7.
URL http://dx.doi.org/10.1007/978-3-540-76440-3_7

[47] F. Barbier, S. Eveillard, K. Youbi, O. Guitton, A. Perrier, E. Cariou, Model-Driven Reverse Engineering of COBOL-Based Applications, in: Information Systems Transformation, Elsevier Inc., 2010, pp. 283–299. doi:10.1016/B978-0-12-374913-0.00011-1.

[48] C. Blanco, R. Pérez-Castillo, A. Hernández, E. Fernández-Medina, J. Trujillo, Towards a modernization process for secure data warehouses, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2009, pp. 24–35. doi:10.1007/978-3-642-03730-6_3.

[49] H. Bruneliere, J. Cabot, G. Dupé, F. Madiot, Modisco: A model driven reverse engineering framework, Information and Software Technology 56 (8) (2014) 1012 – 1032. doi:http://dx.doi.org/10.1016/j.infsof.2014.04.007.

[50] E. Sosa-Sanchez, P. J. Clemente, M. Sanchez-Cabrera, J. M. Conejero, R. Rodriguez-Echeverria, F. Sanchez-Figueroa, Service Discovery Using a Semantic Algorithm in a SOA Modernization Process from Legacy Web Applications, in: SERVICES conf., 2014, pp. 470–477. doi:10.1109/SERVICES.2014.90.

[51] R. Fuentes-Fernández, J. Pavón, F. Garijo, A model-driven process for the modernization of component-based systems, in: Science of Computer Programming, 2012, pp. 247–269. doi:10.1016/j.scico.2011.04.003.

[52] A. Fuhr, T. Horn, V. Riediger, A. Winter, Model-driven software migration into service-oriented architectures, Computer Science-Research and Development 28 (1) (2013) 65–84.

[53] C. Wagner, Model-driven software migration, in: Model-Driven Software Migration: A Methodology, Springer, 2014, pp. 67–105.

[54] F. Bermúdez Ruiz, J. García Molina, O. Díaz García, On the application of model-driven engineering in data reengineering, Information Systems 72 (2017) 136–160. doi:10.1016/j.is.2017.10.004.

10. Annex

Table 6. Secondary pattern catalogue.

Pattern Name	Purpose	Pseudocode	Parameters
		Field Validation	
CMP_-VAL; Greater, Less, Equal than validation.	It Validates if a field value is greater/-less/equal than/to other field.	IF(fieldA [> | >= | <= | < | =] fieldB): SHOW MESSAGE(msg); RAISE ERROR;	- fieldA, fieldB: The values to compare. - msg: The message shown to the user.
ENU_-VAL; Validate enumeration .	It Validates if a field value exists in a pre-defined list of values.	IF NOT(field = literal1 or field = literal2 or ...): SHOW MESSAGE(msg); RAISE ERROR;	- field: The value to validate. - literal1, literal2, ...: The value enumerations. - msg: The message shown to the user.
CPX_VAL; Complex validation.	It validates if a condition occurs, this involves fields, literals, logic and arithmetic operators.	IF (condition): SHOW MESSAGE(msg); RAISE ERROR;	- condition: The expression to validate. - msg: The message shown to the user.

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
NUL_VAL; Null validation.	It validates if a mandatory field has been filled up by the user.	IF (field IS EMPTY): SHOW MESSAGE(msg); RAISE ERROR;	- field: The value to be validated. - msg: The message shown to the user.
RGN_VAL; Range validation.	It validates if the field value is in a range of possible values.	IF (field < minValue and field > maxValue): SHOW MESSAGE(msg); RAISE ERROR;	- field: The value to be validated. - minValue, maxValue: The lower/upper bounds of the range. - msg: The message shown to the user.
		Field Population	
POP_SEQ; Populate a field with an Oracle sequence value.	It get the next value of a sequence and assign it to a field.	SELECT sequenceName.nextval INTO field; FROM dual;	- field: The field to populate. - sequenceName: The Oracle database sequence.

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
POP_INP; Populate a field with the sequence value set in a master table.	Given a detail record, get the value of a sequence set in a master record and assign it to a field; the sequence increases by 1.	SELECT max (*col*0) + 1 INTO field; FROM tableName; WHERE col1 = fieldP AND col2 = field Q AND ...;	- field: The field to populate. - tableName: Detail table. - fieldP, FieldQ, ...: Fields defining the foreign keys. - col0: Column used as a sequence in the Master table. - col1, col2, ...: Foreign keys referring to the master table.
POP_DFL; Populate a field with a default value.	It populates a field with a default value when the users not provide such value.	IF (field IS EMPTY): field ← defaultValue;	- field: The field to populate. - defaultValue: The expression value assigned to the field.

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
POP_SEL; Populate fields with a query result.	It populates fields with the result of a SELECT statement.	SELECT col1, col2, ... INTO fieldA, fieldB; FROM tableName; WHERE col5 = fieldP AND col6 = fieldQ AND ...; IF(no_data_retrieved): SHOW MESSAGE(msg); RAISE ERROR;	- fieldA, fieldB, ...: The fields to be populated. - tableName: The name of the table the data comes from. - col1, col2, ...: The columns of the table to be retrieved and assigned to fields. - col5, col6, ...: These columns help to extract only the records that fulfill a criteria. - fieldP, fieldQ, ...: The expected values. - msg: The message to be shown to the user.

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
POP_-FUN; Populate a field with a function's returned value.	It populates a filed with the result of a function or a literal value.	field ← functionName(param1, param2, ...);	- field: The field to be populated. - functionName: The called function. - param1, param2, ...: The expressions which are passed as arguments to the function.

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
		Model Constraints	
UNQ_VAL; Unique key validation.	It validates if a database table contains a record that coincide with an specific value.	SELECT count(1) INTO localVar; FROM tableName; WHERE col1 = fieldA AND col2 = fieldB AND ...; IF(localVar > 0): SHOW MESSAGE(msg); RAISE ERROR;	- tableName: The database table. - col1, col2, ...: Columns to help to filter the records that coincides with expected values. - fieldA, fieldB, ...: The expected values. - msg: The message shown to the user.
SYN_DET; Master-Detail synchronization.	It retrieves the detail records when the current master is changed.	IF (masterPK IS NOT EMPTY): REFRESH DETAIL;	- masterPK: Master's primary key.

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
DEL_VAL; Delete validation in Master-Detail relationship.	It validates if there is any detail record when the user is trying to delete a master record.	CURSOR_DECLARATION cursor as SELECT 'X' FROM tableName WHERE col1 = fieldA AND col2 = fieldB AND ... FETCH_RECORD cursor INTO localvar IF(recordFound): SHOW MESSAGE(msg) RAISE ERROR	- tableName: Detail table. - fieldA, fieldB, ...: The primary key of a master record. - col1, col2, ...: Foreign key of detail table.
EXT_VAL; Foreign key validation.	It validates if a parent record exists.	SELECT count(1) INTO localVar FROM tableName WHERE col1 = fieldA AND col2 = fieldB AND ... IF localVar = 0: SHOW MESSAGE(msg) RAISE ERROR	- tableName: Master table. - fieldA, fieldB, ...: Foreign key of detail table. - col1, col2, ...: Primary key of master record. - msg: The message to be shown to the user.

Miscellaneous

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
LOGG; Logging.	It records data about the user transactions.	INSERT INTO tableName VALUES (expression1 , expression2, ...);	- tableName: Logging table. - expression1, expression2, ...: Values stored as logs records.
TEX_MSG; Text message management.	It translates codes and system errors into user friendly messages.	IF (MESSAGE CODE = errorCode1): SHOW MESSAGE (messageText1); IF(MESSAGE CODE = errorCode2): SHOW MESSAGE (messageText2);	- procedureName1, procedureName2, ...: The called procedures. - param11, param12, param21, param22, ...: Expressions passed as arguments to the procedures.

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
USR_OPT; User options.	It prevents the user from executing an insert, update or delete operation.	IF (userAction IS [insert | update | delete]): SHOW MESSAGE(msg); RAISE ERROR;	- userAction: The operation or transaction that the user is trying to execute. - msg: The message shown to the user.
CLR_FRM; Clear a Form.	It clears the fields in a Form.	CLEAR FORM;	None

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
FRM_-PRP; Oracle Forms propietary routines.	Oracle Forms offers a rich programming API to control the user interface and access local resources (e.g., get mouse position, move/hide windows, show the menu, etc). This pattern represents triggers that use only this kind of statements.	forms_proprietary_procedure();	None

Continued on next page

Table 6 – continued from previous page

Pattern Name	Purpose	Pseudocode	Parameters
PRC_INV; Procedure invocation.	It calls one or multiple procedures.	CALL procedureName1(param1, param2, ...) CALL procedureName2(param11, param12, ...)	- procedureName1, procedureName2, ...: Procedures to be called. - param1, param2, param11, param12, ...: Expressions which are passed as arguments to the procedures.

Appendix .1. Response Letter

Dear Editors and Reviewers,

Thank you very much for your second round remarks. They helped us to further improve the article. Please find below our answers to your comments. We hope they can help clarify some remaining issues regarding our article.

As required by the reviewing process, we are submitting this response letter along with a revised version of our paper entitled "Model-based Assisted Migration of Oracle Forms Applications: The Overall Process in an Industrial Setting".

In this paper we aim at reverse engineering the PL/SQL code embedded in Oracle Forms applications, with a program understanding intend. The difference with respect to previous works is that our approach goes beyond the syntactic level: it specifically targets the semantics of this domain by first grouping PL/SQL statements that are typically used together. Furthermore, we aim to discover business-level functionality regardless of the number and complexity of the statements required to perform it.

In addition, to tackle the challenges related to migration project management, on top of the legacy/intermediate models, our approach has functional features to help developers to estimate, plan, and track the project progress.

There are 3 remarks for this iteration of the paper. Each remark is followed by our response.

The remarks with specific changes in the paper have a reference to the location in the new paper version where the solution is found. To this end, either we refer to the section where a given modification was introduced or we use an icon in the paper margin for easy identification. Additionally, in the margin we point to the specific remark the modification is addressing.

In what follows we describe how we have addressed the reviewer comments in detail.

Appendix .2. Review #1

Remark 1: I still find the explanation fo the introduction of their own PIM starting in Sec 4 on page 20 line 47 a bit fuzzy. What does "tough" or "confusing"? The point for developing a more simple model (less concepts, better learning curve, domain focussed, ...) can be made in a more scientific way.

Response: Given our interest in smoothing the appropriation of the PIM among practitioners (i.e., a better learning curve), who are not necessarily MDE experts, we designed an intermediate metamodel that has less concepts than standard metamodels (e.g., KDM) and focuses on the domain of 4GL technologies. We made this decision based on previous experiences where we observed two disadvantages in the use of KDM: i) it is tough for developers to appropriate the KDM concepts because the metamodel is quite large; ii) some concepts are too abstract, leading developers to different comprehension and application of those concepts, creating confusion in the team. For example, they tend to map the same legacy concept to different KDM concepts that look similar at first glance. In the margin of section 4, we point to the modification that addresses this remark by using remark number 1.

Remark 2: "5.1.2. How was the catalog built?" should be renamed to "Building the Pattern Catalog" or similar to stay consistent with other subsection names.

Response: We modified the title as indicated by Remark 2.

Remark 3: There are some minor formatting issues, e.g. Fig 6 and Tab 1 placed on quite empty pages. Table 6 in the appendix is a pain to read because of the narrow columns. This could be formatted in some different way, or, maybe, the relatively pointless pseudocode can be removed.

Response: We relocated Fig 6 and Tab 1. In addition, we increased the width of the pseudocode column in Tab 6.