Developing a Model-Driven Reengineering Approach for Migrating PL/SQL Triggers to Java: A Practical Experience

Carlos Javier Fernández Candel[a], Jesús García Molina[a], Francisco Javier Bermúdez Ruiz[a], Jose Ramón Hoyos Barceló[a], Diego Sevilla Ruiz[a], Benito José Cuesta Viera[b]

[a]*Faculty of Informatics, University of Murcia, Spain*
[b]*Open Canarias S.L., Spain*

Abstract

Model-driven software engineering (MDE) techniques are not only useful in forward engineering scenarios, but can also be successfully applied to evolve existing systems. RAD (Rapid Application Development) platforms emerged in the nineties, but the success of modern software technologies motivated that most of the RAD environments were discontinued. Enterprises then had to tackle the migration of RAD applications, such as Oracle Forms. Our research group has collaborated with a software company in developing a solution to migrate PL/SQL monolithic code on Forms triggers and program units to Java code separated in several tiers.

Our research focused on the model-driven reengineering process applied to develop the migration tool for the conversion of PL/SQL code to Java. Legacy code is represented in form of KDM (Knowledge-Discovery Metamodel) models. In this paper, we propose a software process to implement a model-driven re-engineering. This process integrates a TDD-like approach to incrementally develop model transformations with three kinds of validations for the generated code. The implementation and validation of the re-engineering approach are explained in detail, as well as the evaluation of some issues related with the application of MDE.

Keywords: Software Modernization, Reengineering, KDM, Oracle Forms, Model-driven Software Modernization, Model-driven Development

1. Introduction

Model-driven software engineering (MDE) techniques are not only useful in forward engineering scenarios, but can also be successfully applied to evolve existing systems. Models are very appropriate to represent information involved in evolution tasks at a higher level of abstraction (e.g. information harvested into a reverse engineering process). Moreover, model transformation technologies have been developed to facilitate the automation of such tasks.

RAD (Rapid Application Development) platforms emerged in the nineties to offer an agile way of developing software. Agility was the result of applying a GUI-centered paradigm in which most application code was entangled in the event handlers. Therefore, the gain of productivity was achieved at the expense of software quality, and maintenance was negatively affected. The success of the object-oriented paradigm and the appearance of modern software technologies motivated most of the RAD environments to be discontinued. Enterprises then had to tackle the migration of RAD applications. Oracle Forms has been one of the most widely used RAD technologies over last three decades. Although the Oracle company continues supporting and offering solutions to integrate Forms with new technologies (e.g. Web and Java), many enterprises around the world have migrated its Forms legacy code to modern platforms in order to take advantage of new software technologies.

Open Canarias is a Spanish software company with years of experience in software modernization, specially in the banking area. In 2017, Open Canarias launched the Morpheus project[1] to develop a tool able of achieving the highest degree of automation possible in the migration of Oracle Forms applications to object-oriented platforms. An MVC architecture based on Java frameworks was the selected target platform.

Our research group has collaborated with Open Canarias in developing a solution to migrate PL/SQL code on triggers and program units to Java code. These triggers mix concerns that are separated in several tiers in modern applications. This separation of concerns was one of the main challenges to be tackled. This paper is focused on the model-driven re-engineering process applied to develop this migration tool.

A software migration is a form of modernization in which an existing application is moved to a new platform that offers more benefits (e.g. better maintainability or new functionality.) A re-engineering strategy is commonly applied to migrate sys-

Email addresses: `cjferna@um.es` (Carlos Javier Fernández Candel), `jmolina@um.es` (Jesús García Molina), `fjavier@um.es` (Francisco Javier Bermúdez Ruiz), `jose.hoyos@um.es` (Jose Ramón Hoyos Barceló), `dsevilla@um.es` (Diego Sevilla Ruiz), `bcuesta@opencanarias.es` (Benito José Cuesta Viera)

[1]This project has been partially supported with funds of the Spanish Industry Ministry through the CDTI project: **IDI-20150952** (`http://www.cdti.es/`).

tems in a systematic way [1, 2]. This strategy consists of three stages. Firstly, a reverse engineering state is carried out to obtain a representation of the legacy system at a higher level of abstraction. This representation is mapped to the new architecture in a second stage. Finally, a forward engineering stage generates the target artefacts of the new application from the representation generated in the restructuring stage [2]. Models and model transformations can be used to implement the transformation processes involved in these three stages as illustrated in [3, 4]. Actually, model-driven re-engineering and model-driven reverse engineering are common application scenarios of MDE techniques [5]. Some of the most significant efforts made in model-driven software modernization can be found in a recently published systematic literature review on model-driven reverse engineering approaches [6].

In Morpheus, we have devised a model-driven reengineering process for the conversion PL/SQL to Java. PL/SQL legacy code is represented in form of KDM (Knowledge-Discovery Metamodel) [7] models. To maintain the process at the highest level of abstraction possible, we have used the notion of *idiom*, proposed in a previous work of our group [8]. Idioms are primitive operations commonly used in writing event handlers of RAD applications. In our case, idioms models have been obtained from KDM models, and used to generate a model that represents how legacy code is separated into the three tiers of an MVC architecture. In turn, these target platform models are transformed into Object-Oriented models before generating Java code of the new application.

We have defined a development process that involves two main stages. First, individual model transformations are incrementally implemented following a strategy based on unit tests. Once all the transformations have been completed, several kinds of testing are applied on the generated code. The migration tool has been validated by means of forms provided by Open Canarias, which are part of real applications. In this paper, we will describe in detail each stage of the re-engineering solution proposed and the validation process carried out. For each model transformation, we will discuss how it has been implemented and tested. A trigger example will be used to illustrate how each transformation works. Moreover, we evaluate some key aspects of a MDE solution in the migration scenario here considered such as (i) the adoption of KDM; (ii) the choice of the language for writing model-to-model transformations; (iii) the difficulties encountered for testing transformations. Moreover, we will propose a model visualization technique based on the conversion of models into graph database instances.

Contributions. Model-driven reengineering approaches for migrating monolithic legacy code to 3-tier architectures have been presented in [8] and [9]. These works are mainly focused on the reverse engineering stage Moreover, they do not pay attention to the development process applied to implement the re-engineering strategy. In particular, how each model transformation and the final solution were validated is not addressed.

Writing model transformations is recognized as a challenging task in developing model transformations [10], and some approaches to develop transformation chains [11] or individual transformations [12] have been proposed. However, experiences on real projects are not published as far as we know. Test-driven development is also an idea not yet sufficiently explored for individual transformations, although it has been proposed for incrementally developing transformation chains [11].

A limited number of industrial experiences of model-driven modernization have been reported so far. This is evidenced by the low number of model-driven reverse engineering works obtained in the systematic literature review presented in [6]. Moreover, only three of the eleven papers considered in that review used the standard KDM metamodel. To our knowledge, no work on a source-source conversion based on KDM has been published so far.

Several commercial and open source tools are available to migrate PL/SQL triggers to the Java platform, such as *Ispirer MnMTK* [2] and *PLSQL2Java* [3] These tools convert PL/SQL code into Java, but the separation of tiers is not considered.

With these premises, we introduce the main research contributions of this work:

- A development process for implementing a re-engineering is defined and applied. All the stages of a re-engineering are addressed.
- A test-driven strategy to incrementally develop model transformations in a software migration is proposed.
- A comparison between Java and two widely used transformation languages (ATL [13] and QVT operational [14]) is presented.
- A novel strategy for visualizing models in graph databases to help with testing is shown.
- KDM is used and evaluated in a scenario of code migration.

It should also be noted that a valuable contribution of our work is the ability to develop a model-driven migration solution to be integrated in a tool developed by a company, which covers the design, implementation, and validation stages as well and evaluation of the practical experience.

Paper organization. Section 2 introduces the KDM metamodel. Section 3 states the requirements and challenges in building the Code Migrator Code. Section 4 outlines the designed re-engineering strategy. Section 5 explains the development process. A trigger example that is used to illustrate how each model transformation works is presented in Section 6. Section 7 explains how each stage of the re-engineering process has been implemented and tested, and describes all the metamodels involved. Section 8 explains how the tool has been validated. Section 9 evaluates some MDE-related issues from the experience gained in our work. Section 10 discusses how our work is related to other proposals. Section 11 draws some conclusions and comments on future works.

[2]http://www.ispirer.com.
[3]https://pitss.com.

2. KDM metamodel

In this Section, we shall introduce the basis of KDM needed for understanding the injection process and the model transformations involved in the reverse engineering stage. First, we will briefly describe how KDM is organized in packages and layers. Next, the Action and Code packages will be explained in more detail.

2.1. KDM overview

ADM is an OMG initiative to support model-driven modernization. ADM was launched in 2003 with the main purpose of providing a set of metamodels aimed to represent the information commonly used in software modernization tasks. KDM is the core metamodel of ADM: it permits the representation of legacy software assets at different levels of abstraction, ranging from source code to higher-level abstractions such as GUI events, platforms, or business rules. KDM models are built from abstract syntax tree (AST) models which conform to the ASTM (Abstract Syntax Tree Metamodel) [15] metamodel. The rest of metamodels and specifications of ADM are based on KDM, such as SMM [16] for representing metrics, and AFP [17] for automating the function points counting.

KDM is a large metamodel, organized in twelve packages which relate to each other. These packages are partitioned in four layers that represent different domains of a software system. The *Infrastructure* layer includes the `Core`, `KDM` and `Source` packages, which provide the basic concepts that are used in the rest of levels. The *Program Elements* layer defines constructs for commonly used programming languages: organizational and descriptive elements (e.g., types, modules, classes, and procedures) in the `Code` package, and behavioral elements (e.g., statements and control flow) in the `Action` package. The *Resource* layer includes packages to describe resources managed at runtime, in particular data, user interfaces, events, and platforms. Finally, the *Abstraction* layer deals with the architectural view, the domain conceptual modeling, and the build process of the system. Notice that the information related to Infrastructure and Program Elements can be directly extracted from the source code, but models for the other packages must be inferred from such information.

The KDM specification also offers an *extension mechanism*, where an extension is defined as a *family of stereotypes*. Each stereotype has a name and aggregates a set of *tags* (pairs name-type) that express the properties or attributes characterizing it. KDM is a language-independent specification, but modeling source code requires to represent the precise semantic of each statement. For this, KDM provides a set of micro-actions, named *Micro-KDM*, which can be thought of as equivalent to an intermediate representation. These micro-actions provide precise semantics for the basic operations in general programming languages, or GPLs, such as comparison, control, and operations on primitive types. Instead of using Micro-KDM, a creator of KDM models for a particular GPL could define a stereotype family, but then these models could not be interchanged with other KDM-based tools.

KDM emerged as a common interchange format to favor interoperability and data exchange among modernization tools developed by different vendors. Three levels of conformance are defined in the KDM specification. Each level establishes the packages that a tool should support to be compliant with that level. A tool is *L0 compliant* if it supports packages in Infrastructure and Program Elements layers. If a tool is L0 compliant and also support a package included in the Resources and Abstractions layers, then it is *L1 compliant for the corresponding package* (e.g. `Data` or `UI`). Finally, L2 level requires to be compliant with all the packages that are part of the Resources and Abstractions layers.

2.2. Code and Action packages

The KDM injector developed by Open Canarias is L1 compliant for the `Data` package, as will be indicated in Section 7. Therefore, the KDM models extracted–to be reverse engineered–will be expressed in terms of elements of the `Code`, `Action`, and `Data` packages. Next, we will describe the essentials of the `Code` and `Action` package, and the `Data` package will be introduced in the following Section.

The `Action` package is defined by extending and referring to elements in the `Code` package. Figure 1 shows how the `Action` package is based on the `Code` package.

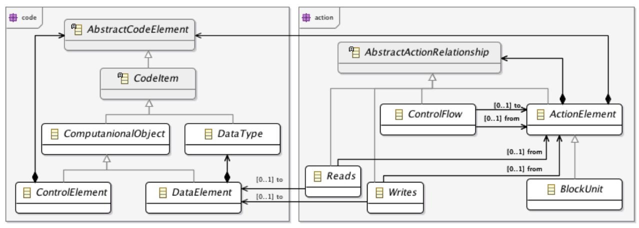


Figure 1: Relationship Between the `Code` and `Action` Packages in KDM.

An excerpt of the Code metamodel appears in Figure 2. `CodeItem` is the central element of this metamodel. It represents any element of the source code. `Module` is the root of the class hierarchy representing units used to package programs (e.g. packages and components). `ComputationalObject` is the root of the `DataElement` and `ControlElement` hierarchies, that represent data and control elements, respectively. `ControlElements` represent callable elements (`CallableUnit`) such as procedures and methods, and `DataElement` represents data items such as variables (`StorableUnit`), parameters (`ParameterUnit`), and literal values (`Value`). Finally, this metamodel also includes elements to define data types (primitive, enumerated, composite, and derived.)

The Action package represents behavior through statements, conditions, code flows, exceptions, and data readings and writings. An excerpt of the Action metamodel appears in Figure 3. `ActionElement` is the central element of this metamodel. Both `ActionElement` and `CodeItem` inherit from the abstract class `AbstractCodeElement`, included in the Code package. `ActionElement` is provided to describe the basic unit of behaviour. An `ActionElement` aggregates instances of classes

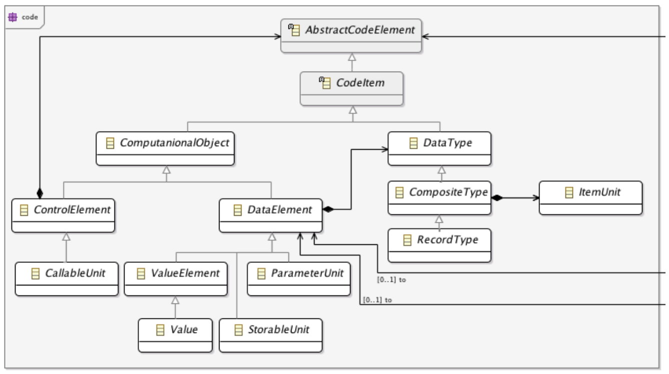


Figure 2: Code package in KDM.

that define relationships whose source is an `ActionElement`. `AbstractActionRelationship` is the root of the class hierarchy that represents relationships such as `ControlFlow`, `Write`, and `Read`. An `ActionElement` object can aggregate instances of these relationship classes, and each relationship instance will have a reference to another `ActionElement` object that describes the code block where the execution flow will continue. Some of the subclasses of `ActionElement` are the following: (i) `BlockUnit` represents a block of `ActionElements`, for example a block of statements in a container as a method or trigger; (ii) `ExceptionUnit` is the root of classes for elements of an exception handler (`TryUnit`, `CatchUnit`, and `FinallyUnit`.)

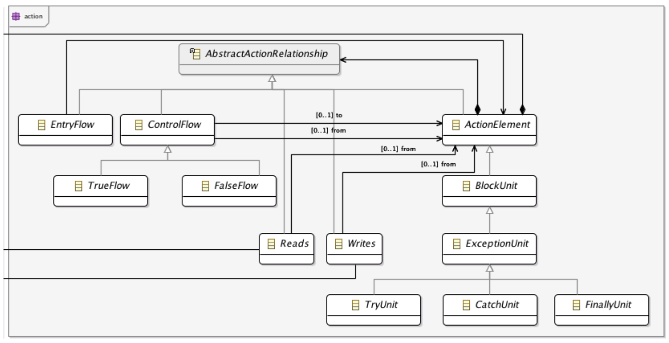


Figure 3: Action Package in KDM.

2.3. *UI package*

The UI package contains elements that represent several aspects of the user interface such as the view structure, the data flow, and the events triggered from visual components. These package depends on the Code and Action packages commented above. Figure 4 shows the elements that represent UI resource: screen, report, field, event, and action. Because the UI elements provided to represent a view structure (layout and kinds of visual components) and event handling are limited, Open Canarias created two family of stereotypes for this package. The UI structure has been represented by means of stereotypes based on the USIXML metamodel [18]. On the other hand, UI interaction is modeled by defining stereotypes based on the IFML metamodel [19].

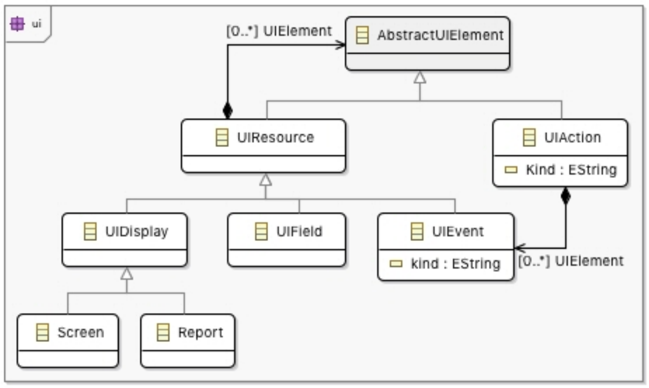


Figure 4: UI Package in KDM.

3. Code Migrator tool: Requirements and Challenges

In Morpheus, Open Canarias organized the migration tool into two main components that correspond to the two main artifacts of an Oracle Forms application: GUIs and event handlers. We refer to these components as *GUI Migrator tool* and *Code Migrator tool*, respectively. Our research group has collaborated in the implementation of the Code Migrator tool, which aims to automate the migration of program units and triggers in Oracle Forms to an MVC Java platform. We have not only considered the building of this tool as a research problem, but also the process defined to develop it. In this section, we shall precisely identify the requirements to be satisfied and the challenges to be addressed in building the tool. In the following sections, the model reengineering strategy designed to carry out the migration and the process defined to implement this strategy will be presented.

As depicted in Figure 5, the input of the tool is the source code and the database schema of the Forms application to be migrated. The output is made up of artifacts generated for a MVC architecture. In the current implementation, the MVC architecture is based on Java frameworks: JSF [20] on the view layer, Spring[4] on the business logic layer, and JPA [21] on the persistence layer.

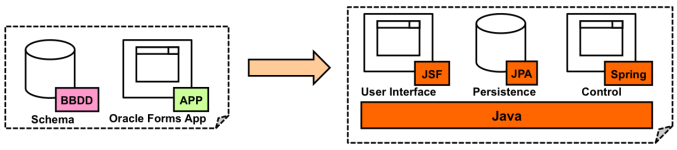


Figure 5: A tool for migrating Oracle Forms applications to Java.

The requirements for the tool are:

[4]https://spring.io/.

- It should translate PL/SQL procedural code into Java code. Because Forms PL/SQL triggers are monolithic, the code generated must be divided into several methods according to the tiers of the target platform.

- *Platform-independence* should be achieved in the design of the tool. It should be designed to facilitate generating code for any object-oriented target platform, e.g., C# for .NET.

- A level of automation close to 100% should be achieved on migrating PL/SQL triggers and program units to Java. The level of automation will be measured by the percentage of legacy PL/SQL code translated to Java.

- Those PL/SQL built-in functions/procedures that are not provided in Java will be manually migrated as part of a library available for all the migration projects.

- The code generated should be integrated into the component that automates the GUI migration.

- The trigger code should not only be migrated, but also the code dealing with database access, and the code of procedures and functions invoked from triggers as program units.

Next, we will identify the main issues to be addressed to satisfy the requirements above exposed.

PL/SQL-to-Java mapping. A mapping between the PL/SQL and Java languages must be established. To achieve a level of 100% automation on the PL/SQL code injected this mapping should cover all the PL/SQL elements: statements, expressions, data types, cursors, exceptions, and collections. Built-in functions and predefined exceptions are two examples of PL/SQL elements for which a direct translation is not possible. In the case of built-in operations, it must be checked whether Java provides equivalent operations or not. They should be manually implemented if they do not exist in Java. Regarding exceptions, Oracle Forms provides a high number of platform-specific exceptions that are not present in the Java platform. This problem should be addressed by defining equivalent exceptions for the target Java platform and omitting those not applicable.

Separation of concerns. As indicated above, event handlers of RAD applications mix code of different aspects of an application (data access, control logic, and business logic.) Conversely, a *separation of concerns* is a key architectural element in modern paradigms. Disentangling the code of PL/SQL triggers is undoubtedly the main challenge in implementing our tool. The code must be analyzed to apply a separation according the tiers or layers of the target platform, in our case, an MVC architecture. The code is separated in fragments which must be categorized as belonging to a particular tier of the target architecture. Once categorized, the code fragments can be translated and included in a method of a tier.

Sharing variables among generated methods. A local variable of a PL/SQL function/procedure must be shared among the Java methods generated for such a function/procedure. A possible solution would be pass values as arguments in method invocations, but arguments are always passed by value in Java, so a different solution must be devised.

Target platform independence. An Object-oriented metamodel would facilitate adapting the solution to different languages. On the other hand, an MVC architecture metamodel would provide independence of a particular target platform.

Integrating GUI and event handler code generated. In our case, we had to address the issue of integrating the code generated for PL/SQL triggers and unit programs with the code previously generated by the GUI Migrator tool.

4. A model-driven reengineering process for the Code Migrator tool

Software reengineering offers a disciplined way to migrate a legacy system [1]. Three kind of activities are normally performed in a reengineering: reverse engineering, forward engineering, and restructuring [22]. Figure 6 shows the horseshoe model [2] that is frequently used to illustrate how reengineering approaches work. Reverse engineering is applied on the existing system in order to harvest knowledge in form of abstract representations. Through a chain of vertical transformations, descriptions of several system aspects are extracted and represented at different levels of abstraction. The new source code is created by means of vertical transformations (forward engineering) whose input are representations defined for aspects of the target system. These representations are obtained through horizontal transformations (restructuring) whose input is a representation at the same level of abstraction produced in the reverse engineering process. Tasks in a reengineering process can be automated, semi-automated, or manual. In particular, transformations can be either manually carried out by developers, or automatically executed by means of specific programs.

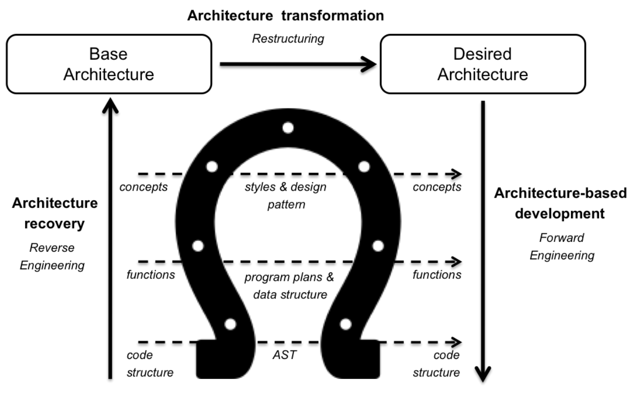


Figure 6: Horseshoe model for reengineering.

Creating abstract representations and writing transformations between representations are, therefore, essential activities in reengineering processes. This has motivated the emergence of model-driven reengineering approaches, which take advantage of MDE techniques, specially (meta)models and model transformations, in order to automate such processes. Metamodels are used to define the different representations, and vertical and horizontal transformations are automated with model transformations: text-to-model transformations to inject code into models, model-to-text transformations to generate code from models, and model-to-model transformations when the input and output are models.

Figure 7 shows the horseshoe model for the model-driven reengineering process defined for the Code Migrator tool. Forms PL/SQL triggers are moved to a MVC architecture based on Java frameworks through a model transformation chain. A reverse engineering based on the approach proposed in [8] has been applied to harvest an architectural representation adequate to migrate PL/SQL triggers to object-oriented languages. That approach proposes to express behavior of event handlers in terms of primitive operations (i.e. code patterns) commonly used in RAD applications (e.g. read data from a database or write data in GUI controls.) Such primitive operations are referred to as *idioms*. Our solution differs from [8] in two significant aspects: (i) Code and data are abstracted in form of KDM models, instead of using AST models for the event handler code; and (ii) the strategy used to separate the legacy code in several concerns in the final application, as explained in Section 7. Benefits provided by KDM will be commented in Section 9. In addition, we had to extend the set of idioms considered in [8] in order to be able to cover nearly 100% of the PL/SQL statements. Figure 13 shows the metamodel defined to describe the PL/SQL code in terms of idioms. As shown in Figure 7, an Idiom model is obtained in two steps. First, code is injected into KDM models, i.e. using a text-to-model (t2m) transformation. Next, a m2m transformation named *kdm2idioms* analyzes the KDM model and creates the corresponding Idiom model.

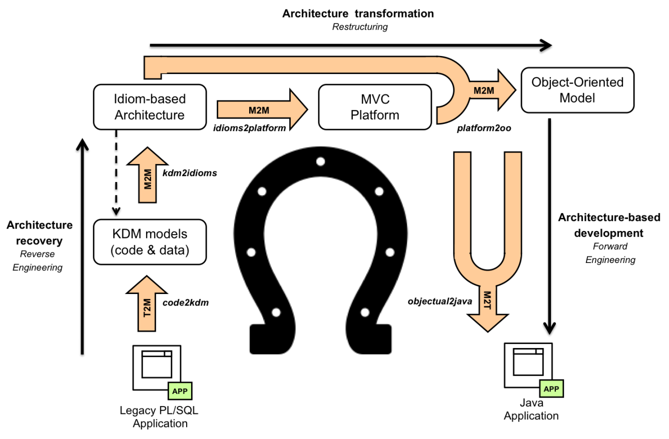


Figure 7: Code reengineering process in Morpheus.

An architectural transformation or restructuring stage converts the Idiom model into a Platform model that represents how the code is organized in the target platform required by Open Canarias. For this, we have defined the MVC architecture and Objectual metamodels and a chain of two a model-to-model transformation. Both metamodels are complementary. How the monolithic legacy code is separated into tiers is represented in a MVC architecture model, which has references to elements of the Objectual model that abstract the object-oriented code to be generated. First, a m2m transformation, named *Idioms2Platform*, generates a MVC architecture model from the Idioms model. This transformation discovers which classes and methods must be generated. A second m2m transformation, named *Idioms-Platform2Objectual*, determines which layer each class and method belongs to, and obtains a object-oriented representation for each idiom. This partitioning requires not only information provided by Idiom models but also accessing to KDM elements that are referenced by idioms.

Finally, the MVC architecture and Objectual models are used to apply a forward engineering stage that generates the code that results from the migration. A m2t transformation implements this stage and generates the Java code that corresponds to the PL/SQL legacy code.

Noting that dotted lines have been included in Figure 7 to indicate traceability relationships among models.

5. A development process for the Code Migrator tool

The Code Migrator tool enacts the model-driven reengineering process described in the previous section. The tool consists on the model transformation chain that automates the reengineering approach applied to move Forms PL/SQL code (triggers and program units) to a Java platform. In this Section, we will present the software process followed to develop the tool.

An iterative and incremental software development life cycle has been applied, as shown in Figure 8.

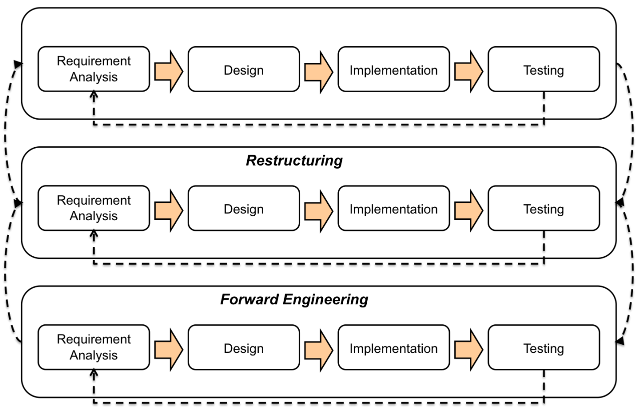


Figure 8: Iterative and incremental life cycle for developing the migration tool.

We established three iterations, one for each stage of the reengineering process, shown in Figure 7. Each iteration involves requirement analysis, design, implementation of a model-driven solution for the corresponding reengineering stage, and testing. Requirements were elicited by the company from their previous experience in manual Forms-to-Java migration.

The greatest effort has been devoted to the implementation and testing of the three stages of the reengineering process. As indicated in Section 4, the injector that converts PL/SQL code into KDM models was developed by Open Canarias. Thus, we tackled the implementation of 3 model-to-model (m2m) transformations and 1 model-to-text (m2t) transformation. Moreover, we had to define the target metamodel for each of the m2m transformations. Testing has been performed for each implemented model transformation, and once the forward engineering stage was completed, a validation of the solution was performed, as shown in Figure 9.

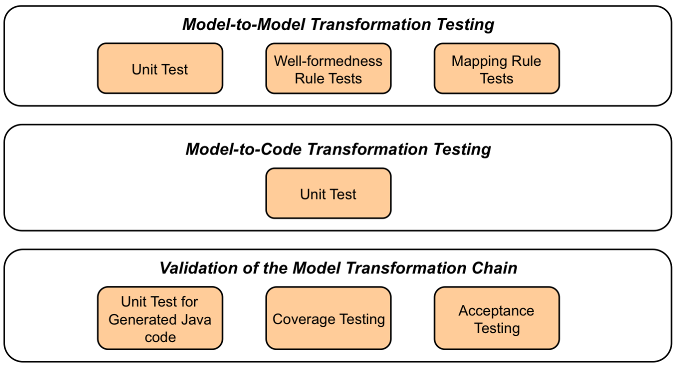

Model-to-Model Transformation Testing
Unit Test
Well-formedness Rule Tests
Mapping Rule Tests
Model-to-Code Transformation Testing
Unit Test
Validation of the Model Transformation Chain
Unit Test for Generated Java code
Coverage Testing
Acceptance Testing

Figure 9: Testing activities in implementing the reengineering process.

Model-to-model transformations are usually complex, specially when the involved metamodels are large and complex. Therefore, both writing and testing m2m transformations are challenging activities. Like GPL programs, m2m transformation definitions must be tested to detect defects in the implementation. A testing process must be applied to assure that each transformation in an MDE solution (i.e. a model transformation chain) properly works.

5.1. Writing model-to-model transformations in Java

As part of the reengineering process we used to develop the tool, we factorized the Java code for model-to-model transformations as much as possible, resulting in the same high-level design. The code was organized in the following components:

1. *Iterator*, that traverses the source model.
2. *Analyzer*, that analyzes the source current element to identify what target elements must be created to resolve a mapping.
3. *Builder*, that creates and initializes the instances of the target model and aggregates these instances to the object that aggregate them.
4. *Reference resolver*, in charge of connecting a created element to existing elements according to the references in the target metamodel.

In addition, a particular transformation can have additional components to implement some specific functionality or utility. The number of classes of a component, as well as its complexity, depends on the characteristics of the transformation.

The Iterator component iterates on the set of root elements of aggregation hierarchies in the source model, and for each one of them traverses all its aggregation paths. For each visited element, the *Analyzer*, *Builder*, and *Reference resolver* components collaborate to create the corresponding target elements. All the m2m transformations will be explained in detail in Section 7. For each transformation, we shall describe how the components works and how the testing has been applied.

5.2. Testing model-to-model transformations

Model-to-model transformation testing has been extensively addressed in the literature. Three main challenges are commonly identified [23, 24, 11]: (i) creating a set of input models (test models) to test the transformation under study, (ii) defining adequacy criteria for checking if test models are sufficient for testing, (iii) checking that the result of a test is actually the expected model. A discussion on these challenges can be found in [23], where some emerging approaches to overcome them are outlined. The source metamodel of a transformation can be used to generate test models automatically. However, this generation is a complex problem because it is difficult to assure that a set of models satisfies all the constraints required for a particular test. Because of this, test models are manually created by using model editors; if there is no editor specially created for a particular metamodel, model management frameworks, such as Eclipse/EMF, offer tooling to automatically create a generic editor for a metamodel. Model comparing tools can be used to check if a transformation produces the expected result for a test model. However, manually or automatically building expected result models is again difficult, so model-to-model transformations are usually validated by checking a set of constraints on the result model in order to determine if mappings have been correctly applied.

Some test-driven development approaches have been proposed to implement m2m transformations. Unit tests have been considered in [25, 26, 27], and an incremental process to test model transformation chains is described in [11], which proposes four kinds of transformations. Actually, there is little consensus in how to effectively test transformations, as it is a very difficult task. Specially when source and target metamodels are large and complex, such as those in our project.

5.3. A test-driven development approach

On developing the Code Migrator tool, we have defined a test-driven approach to write transformations in an incremental way, which is shown in Figure 10. In each step, the methods that resolve a particular mapping are written, and a small input model that only contains instances of the classes involved in the mapping is created. Then, the transformation is executed for this input model. If errors are produced, the methods are fixed. Otherwise, a new mapping is considered, and both the methods implementing it and the test model are created, and the transformation is again executed. Executed methods are also stored in order to be executed again later, along with methods for new mappings. The process continues until all the mappings are implemented, and the transformation is completed. It should be noted that the implementation of a particular mapping involves to add methods to the *Analyzer*, *Builder*, and *Reference resolver* components of the model transformation. *Iterator* usually does not need to be modified.

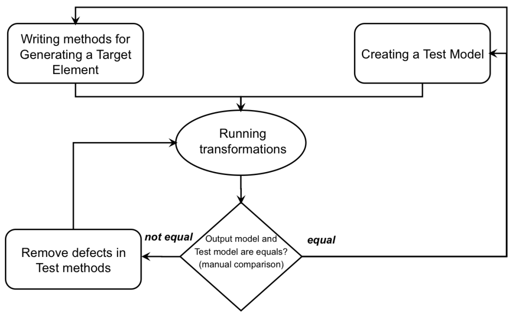

Writing methods for Generating a Target Element
Creating a Test Model
Running transformations
Remove defects in Test methods
not equal
Output model and Test model are equals? (manual comparison)
equal

Figure 10: Incremental process for model-to-model transformations.

Once a m2m transformation is completed following the incremental strategy commented above, a test is applied to check if generated models satisfy a set of constraints, which are of two kinds: (i) a set of well-formedness rules for each metamodel, and (ii) for each transformation, a set of rules to check if each mapping has been correctly applied. Both kinds of rules have been implemented in Java. Checking well-formedness and mapping rules correspond, respectively, to the integrity and invariant tests considered in [11]. As described in more detail in Section 8, we have tested each model-to-model transformations using three test models whose size is considered small, medium, and large. It should be noted that no input model had to be created by hand, as an injector is available to obtain KDM models from PL/SQL code. This saved us a great effort. We have manually checked that injected code for the small, medium, and large test cases include all the PL/SQL statements. Moreover, as it can be seen in Table 1, the high number of triggers, program units and lines of code in these cases assure that each transformation has been sufficiently tested.

Table 1: Triggers, Program Units and LoC of the tested forms

Forms	#triggers	triggers LoC	#units	units LoC	Total LoC
small	5	807	25	442	1249
medium	42	2206	30	1302	3508
large	251	6563	68	4617	11180

Regarding validation of the automatically generated code, three kinds of tests have been applied as shown in Figure 8. First, *unit tests* were written for the generated code for each test case. The output code was executed and it was checked that there are not errors in runtime. A *coverage testing* is also applied. Metrics about number of triggers, program units and sentences have been calculated, as well as metrics for the equivalent software elements on the target platform. Then, it has been checked that each previous Forms element is covered on the Java platform. Finally, an *acceptance test* is applied, in which we execute each Java event handler generated to check that it has the same behavior as the corresponding PL/SQL trigger. The checking is performed manually and Mocks have been provided to complete the testing.

6. An example of trigger migration

Here we shall present a example of trigger and a program unit migration that will be used to illustrate how each of the model transformations shown in the following Section work. First, we briefly introduce the structure of a Forms application and show the trigger example and the program unit. Then, we describe the target architecture and show the Java code that should be obtained for a migration to a platform based on the JSF (Java ServerFaces) and Spring framework.

Form applications.

An Oracle Forms application is composed of one or more forms. Each form contains the artifacts that implement both the user interface and the business and application logic. A form includes a set of visual components (i.e. widgets) that belong to one or more windows. Most of the application code is implemented as triggers that handle visual component events. These triggers are written in PL/SQL, and can be viewed as the typical event handler in similar RAD architectures. Apart from triggers, another Form construct that uses PL/SQL code is a Program Unit. A Program Unit is an auxiliary function or procedure that provides support for implementing business and application logic. They can be invoked from triggers.

A PL/SQL trigger example.

We have defined a simple trigger that includes database access and operations that read or write information on the visual components of a window. This trigger could be part of an application intended to manage grants for companies. It supports the functionality of applying a new grant when funds of a previous grant have been spent. We suppose that this trigger manages the "pressed" event of a button named "New_Grant". A new grant can only be applied if the payments made under the previous grant exceed the threshold indicated in the request. Therefore, the trigger has to read the payments and the threshold value from the database. Then all the payments are added to calculate the paid money total. If this total amount is greater than the threshold, then (i) the old grant is updated, (ii) a new grant is created, and (iii) the result of the operation is shown in an application window. Following, an excerpt of the code implemented by the button trigger is shown:

```
BEGIN
  BEGIN
    company_name := normalize_company_name(:COMPANY
        );
    SELECT sum(PAYMENT) INTO money_paid FROM GRANTS
        .GRANTS_PAYMENTS WHERE ...;
    SELECT threshold, endowment INTO threshold,
        endowment FROM GRANTS.COMPANY_GRANTS WHERE
        ...;

    total := 2 * endowment - money_paid;

    IF money_paid >= threshold THEN
      UPDATE GRANTS.COMPANY_GRANTS_GRANTED SET
          state = 'SUSPENDED' WHERE ...;
      INSERT INTO GRANTS.COMPANY_GRANTS_GRANTED
          (...) VALUES (:GRANT_CODE, company_name
          ...);
    END IF;
  EXCEPTION WHEN OTHERS THEN
    message('Database unaccesible');
```

```
      RAISE FORM_TRIGGER_FAILURE;
    END;

    IF money_paid >= threshold THEN
      SET_ITEM_PROPERTY('...GRANT_RENEWED', visible,
          property_true);
    ELSE
      SET_ITEM_PROPERTY('...THRESHOLD_NOT_EXCEEDED',
          visible, property_true);
    END IF;

    diference := threshold - money_paid;
    IF diference > 0 THEN
      :RENEW_COMPANY_GRANTS.THRESHOLD_DIFERENCE :=
          diference;
    ELSE
      :RENEW_COMPANY_GRANTS.TOTAL_AMOUNT := 2 *
          endowment - money_paid;
    END IF;
END;
```

Note that the trigger code mixes application logic, database access and GUI reading/writing operations. The trigger calls the function `normalize_company_name` that would be part of a Program Unit.

```
FUNCTION normalize_company_name (company_name IN
    VARCHAR2) RETURN VARCHAR2 IS
BEGIN
  IF length(company_name) > 256 THEN return substr(
      company_name, 1, 256); END IF;
  return company_name;
END;
```

In addition, two PL/SQL built-in functions are used: `length()` returns the size of a `Varchar` variable, and `substr()` returns a substring of a `Varchar` variable.

A MVC architecture based on Java technologies.

Next, we will explain the target architecture and show the code generated for the example. This information will help to understand the metamodels and model transformations explained in the following Section.

According to the requirements of the Open Canarias company, the Java application to be generated from the legacy code should be organized in accordance with a MVC architecture based on Java technologies. In particular, the company requires three Java frameworks to be stacked to form the MVC architecture: JSF (Java ServerFaces) for the view layer, Spring for organizing the code of the application logic and supporting the Controller layer, and JPA (Java Persistence API) for providing persistence/access to business entities.

In Spring MVC each controller is annotated by the *@Controller* tag. The navigation among views is given by the use of controllers and their methods annotated as *@RequestMapping*. The application logic is implemented in services classes (also annotated as *@Service*) and they are declared in controllers by the injection dependency technique through the *@Autowired* declaration.

JSF allows declaring the view. It enables programmers to implements the view in two separated artefacts: (i) a *view* that is composed of UI widgets, which is implemented as a JSP or Facelets file, and (ii) a *Managed Bean* that provides view backing for the UI widgets, which is implemented as a Java class. Accessing widget values is achieved through attribute declarations and their corresponding getter/setter methods. Widget events are handled through Java methods of the Managed Bean class. These event handler methods usually contains UI logic that manipulates the widgets in the view, and they delegate the application logic and data access on the application controllers. This separation was required by the Open Canarias company for the applications resulting of the migration process here considered. These applications therefore contain the following elements:

- A JSF Managed Bean class for each window in the input form. This class contains a method for each trigger associated to visual components in the window.
- A Spring service class that implements an application controller. The class contains the methods in which the code of a legacy trigger is divided. These methods implement the business logic, which contains data access and application logic. In addition, this class contains methods for some database operations, such as insert and update.
- A Spring service class is created to hold the methods resulting of the Program Unit migration. There will be one Spring service class for each Forms which will be in charge of providing all the Java methods resulting of the Program Unit migration.

Variables that are present in a trigger must be shared among the methods that result of its migration: a method in the managed bean class and one or more methods in the service class. To achieve this sharing, we have defined a Java map to store the values associated to each GUI attribute. This map contains pairs whose key is the GUI attribute name and the value is the widget value. A managed bean method pass this map to the invoked service methods. Given the lousy semantics of variable scope in PL/SQL, this map is also used to share variable values among methods.

Managed Bean class skeletons are generated by the GUI Migrator component developed by Open Canarias. These skeletons contain method headers and the JSF attributes (declarations and getter/setter methods), but the body is empty. The implementation of these methods must therefore be completed by the Code Migrator tool here addressed. This conveys a integration problem to be tackled in our reengineering-based solution. The service class with the methods corresponding to the Program Unit procedures/functions must also be generated.

Generated managed bean and services classes.

Next, we show the Managed Bean and Service classes created for the trigger example. The class `RenewGrantsManagedBean` contains the field `RenewGrantsService` that registers the only Spring service used, which is annotated as `@Autowired`. This class also contains the `newGrantButtonWhenButtonPressed` method for the only event handler defined in the form. As noted, this method manipulates GUI elements in order to read the values introduced by users or showing values which are calculated or read from database. It calls methods of the `RenewGrantsService` service in order to perform database operations and calculate new UI data.

```
public class RenewGrantsManagedBean {

  @Autowired private RenewGrantsService
      renewGrantsService;

  public void newGrantButtonWhenButtonPressed() {
    Map<String, Object> map = new HashMap<String,
        Object>();
    map.put("year", renewCompanyGrants.getYear());
    map.put("grantCode", renewCompanyGrants.
        getGrantCode());
    renewGrantsService.
        newGrantButtonWhenButtonPressed1(map);
    if ((Double)map.get("moneyPaid") >= (Double)map
        .get("threshold")) {
      setRenewCompanyGrantsGrantRenewedVisible(true
          ); // Generated by UI Migrator component
    } else {
      setRenewCompanyGrantsThresholdNotExceededVisible
          (true); // Generated by UI Migrator
          component
    }
    renewGrantsService.
        newGrantButtonWhenButtonPressed2(map);
    renewCompanyGrants.setThresholdDiference((
        String)map.get("thresholdDiference")); //
        Generated by UI Migrator component
    renewCompanyGrants.setTotalAmount((String)map.
        get("totalAmount")); // Generated by UI
        Migrator component
  }

  // ... Methods generated by UI Migrator component

}
```

The `RenewGrantsService` class implements the Spring service corresponding to the controller. As shown below, this service class contains the two methods in which the trigger code has been divided. The first one accesses the database, performs a computation, and updates the variable map. Then the second one completes the computation of the values that lately are shown on the user interface. To support the database access, two methods have been injected: `writeToDB()` and `readFromDB()`. These methods allow to execute arbitrary SQL sentences defined in the source application. The execution is independent of the quantity of arguments through the JPA technology.

```
@Service
public class RenewGrantsService {

  @Autowired private RenewGrantsAppService
      renewGrantsAppService;
  private EntityManagerFactory emf;

  public void newGrantButtonWhenButtonPressed1(Map<
      String, Object> map) {
    try {
      String companyName = renewGrantsAppService.
          normalizeCompanyName(map);
      map.put("moneyPaid", readFromDB("SELECT sum(
          PAYMENT) FROM GRANTS.GRANTS_PAYMENTS
          WHERE ..."));
      map.put("endowment", readFromDB("SELECT
          endowment FROM GRANTS.COMPANY_GRANTS ..."
          ));
      map.put("threshold", readFromDB("SELECT
          threshold FROM GRANTS.COMPANY_GRANTS ..."
          ));
      Double total = ((2 * (Double)map.get("
          endowment")) - (Double)map.get("moneyPaid
          "));
      if ((Double)map.get("moneyPaid") >= (Double)
          map.get("threshold")) {
        writeToDB("UPDATE GRANTS.
            COMPANY_GRANTS_GRANTED SET state = '
            SUSPENDED' WHERE ...");
        writeToDB("INSERT INTO GRANTS.
            COMPANY_GRANTS_GRANTED (....) VALUES (
            ? , ? , ? , ? , ? )", ...);
      }
    } catch (Exception e) {
      message("Database unaccesible")/* TODO: PL/
          SQL Library Call */;
      throw new FormTriggerFailure();
    }
  }

  public void newGrantButtonWhenButtonPressed2(Map<
      String, Object> map) {
    Double diference = ((Double)map.get("threshold"
        ) - (Double)map.get("moneyPaid"));
    if (diference > 0) {
      map.put("thresholdDiference", diference);
    } else {
      map.put("totalAmount", ((2 * (Double)map.get(
          "endowment")) - (Double)map.get("
          moneyPaid")));
    }
  }
}
```

Below, it is shown the Service class that implements the `normalizeCompanyName()` method that is part of the Program Unit. The generated code is similar to the PL/SQL code. Those functions which could not be mapped are annotated as manual task to be performed. In the example, these functions are `length` and `substr`.

```
@Service
public class RenewGrantsAppService {
  public String normalizeCompanyName(Map<String,
      Object> map) {
    if (length((String)map.get("companyName"))/*
        TODO: PL/SQL Library Call */ > 256) {
      return substr((String)map.get("companyName"),
          1, 256)/* TODO: PL/SQL Library Call */;
    }
    return (String)map.get("companyName");
  }
}
```

7. Developing the Trigger Migrator tool

This section will explain how each stage of the reengineering process has been implemented. As depicted in Figure 11, the proposed reengineering approach consists of a chain of five model transformations.

- (*Rev. Eng.*) The t2m transformation *code2kdm* injects the legacy source code into the KDM models.
- (*Rev. Eng.*) The m2m transformation *kdm2idioms* obtains idioms models from KDM models.
- (*Rest.*) The m2m transformation *idioms2platform* separates monolithic code in the three tiers of the target MVC architecture.
- (*Rest.*) The m2m transformation *platform2oo* uses the Platform, Idioms, and KDM models to generate an Object-oriented model of the code to be generated.
- (*Forw. Eng.*) The m2t transformation *objectual2java* generates code of the final application from the Object-oriented model.

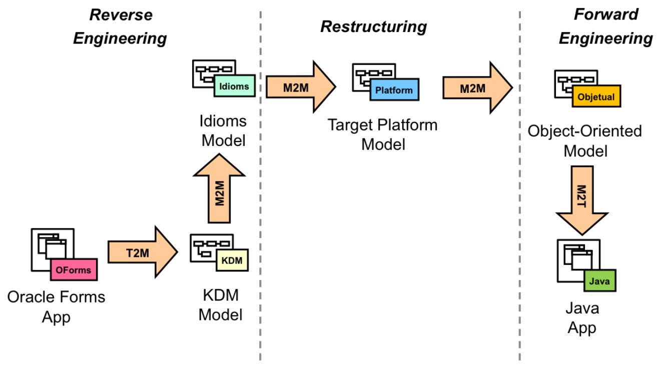


Figure 11: Migration process of the tool.

For each transformation, we will describe how it has been implemented and tested, and the result of being applied to the trigger example. The metamodels involved will be described as they are required. When describing the implementation and testing of the transformations, we will focus on specificities with respect to the explanation given in Section 5.

7.1. Reverse engineering: KDM injection

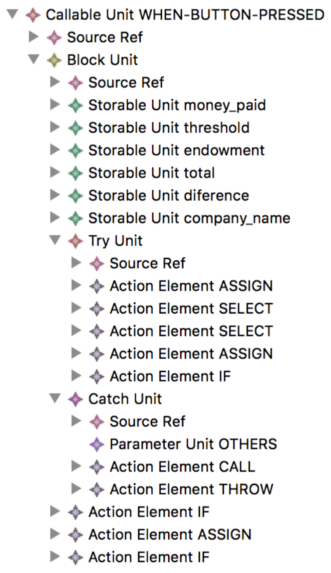
Open Canarias has developed an injector that obtains KDM models from PL/SQL code. As explained in Section 2, ASTM models can be used to generate KDM models. An ASTM model represents source code as an abstract syntax tree. Therefore, a KDM injection process consists of two stages. Firstly, source code is transformed in an ASTM model, and then a model-to-model transformation takes as input the ASTM model and outputs the KDM model. A detailed explanation of this process can be found in [28] for a KDM injector of PL/SQL code.

In order to reuse assets in building KDM injectors for different languages, Open Canarias has also developed the *ModelSET Parser Generator* framework. This frameworks provides support for the tasks involved in implementing a KDM injector: (i) generation of the parser that creates a concrete syntax tree, (ii) transforming the concrete syntax tree into an ASTM model, and (iii) transforming ASTM models into KDM models.

This KDM injector developed by Open Canarias is L1 compliant for the Data package. As explained in 2.3, USIXML and IFML stereotypes have been defined to model UI visual components and interaction, respectively. Therefore, the injection is not L1 compliant for UI. Moreover, the injector has not used Micro-KDM, but a family of stereotypes (e.g. SELECT, IF, CALL, THROW) has been defined to establish the kind of PL-SQL statement represented by an `ActionElement`, as indicated in Section 2. The `kind` and `name` attributes of `ActionElement` are used to indicate the type and name of the stereotype represented, respectively.

7.1.1. Application to the Trigger Example

Figure 12 shows the KDM model injected for the trigger example. An instance of `BlockUnit` represents the only trigger of our example. This instance aggregates (i) a `SourceRef` instance that stores the text of source code, (ii) several `StorableUnit` instances that represent local variables, and (iii) `TryUnit` and `CatchUnit` instances that, in turn, aggregate the stereotyped `ActionElement` elements representing statements in the trigger code. Depending on the kind of sentence, the injector establishes the value to be recorded in the field `name` of an `ActionElement` element as stereotype. In the example, we can see that `ActionElements` with the `ASSIGN`, `SELECT`, `IF`, `CALL`, and `THROW` stereotypes have been created.



```
▼ ✦ Callable Unit WHEN-BUTTON-PRESSED
  ▶ ✦ Source Ref
  ▼ ✦ Block Unit
    ▶ ✦ Source Ref
    ▶ ✦ Storable Unit money_paid
    ▶ ✦ Storable Unit threshold
    ▶ ✦ Storable Unit endowment
    ▶ ✦ Storable Unit total
    ▶ ✦ Storable Unit diference
    ▶ ✦ Storable Unit company_name
    ▼ ✦ Try Unit
      ▶ ✦ Source Ref
      ▶ ✦ Action Element ASSIGN
      ▶ ✦ Action Element SELECT
      ▶ ✦ Action Element SELECT
      ▶ ✦ Action Element ASSIGN
      ▶ ✦ Action Element IF
    ▼ ✦ Catch Unit
      ▶ ✦ Source Ref
        ✦ Parameter Unit OTHERS
      ▶ ✦ Action Element CALL
      ▶ ✦ Action Element THROW
    ▶ ✦ Action Element IF
    ▶ ✦ Action Element ASSIGN
    ▶ ✦ Action Element IF
```

Figure 12: KDM model injected for the trigger example.

This model will be the input of the transformation that complete the reverse engineering stage by obtaining an Idiom model.

7.2. Reverse engineering: Generation of the Idiom model

As indicated in Section 4, our inference process is based on the approach presented in a previous work [8]. The notion of *idiom* was there proposed to achieve a highest level of abstraction. An idiom is a code pattern commonly used by developers of RAD platforms, in our case Forms applications. A Forms

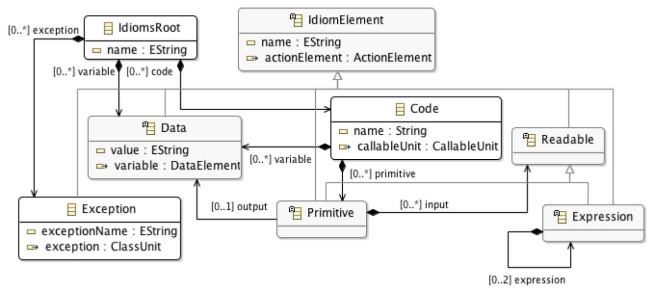


Figure 13: An excerpt of the Idioms metamodel.

trigger and program unit code is represented in terms of idioms to facilitate the migration to the target platform. Table 2 shows some of the idioms defined for PL/SQL.

Next, we will first present the Idioms metamodel, then describe the *kdm2Idioms* transformation that converts KDM injected models into Idioms models, and finally show the idioms model obtained for the trigger example.

7.2.1. Idioms metamodel

Figure 13 shows an excerpt of the Idioms metamodel, in which the hierarchy of classes that represent primitive operations is omitted. The root class of the metamodel is `IdiomsRoot`, which aggregates three kinds of elements: (i) `Code` that represent the code of an event handler, (ii) a set of `Variables` that can have global or local scope, and (iii) a set of `Exceptions` that can be thrown by the application.

`Idiom` is the root class of the hierarchy of classes representing all the idioms defined in our metamodel. Idioms are classified into three categories: `Variable`, `Primitive`, and `Expression`. These classes inherit of `Idiom` and have a reference to the KDM elements from which an idiom was generated. `Primitive`, in turn, is the root of the classes that represent the idioms defined in Table 2.

A `Code` is composed of a set of local `Variables`, and a set of `Primitives`. A `Primitive` aggregates a set of zero or more `Readables` as input, and it can reference to a `Variable` as output. Each`Readable` provides an input value, and the result is stored into a `Variable`. A `Readable` can be a primitive, a variable reference (`VariableRef`), or a value returned by a function call (`ReturnValue`). An `Expression` represents a conditional expression, and they can be nested. Below, the classes that represent expressions are commented.

Figure 14 shows the hierarchy of `Primitive` classes in the Idioms metamodel. `Loop`, `SelectionFlow` and `Try` represent the three more common primitives. They are characterized for being composed of other primitives. A `Loop` aggregates the `Expression` that represents the iteration control condition, and a set of primitives that correspond to the iteration body. `SelectionFlow` represents the selection of an execution flow, either with *if* or *switch* semantics. A `SelectionFlow` is composed of a set of `Case` elements, which are primitives having associated a condition (`Expression` element) and a set of primitives representing the code block to be executed when the expression is evaluated as `true`.

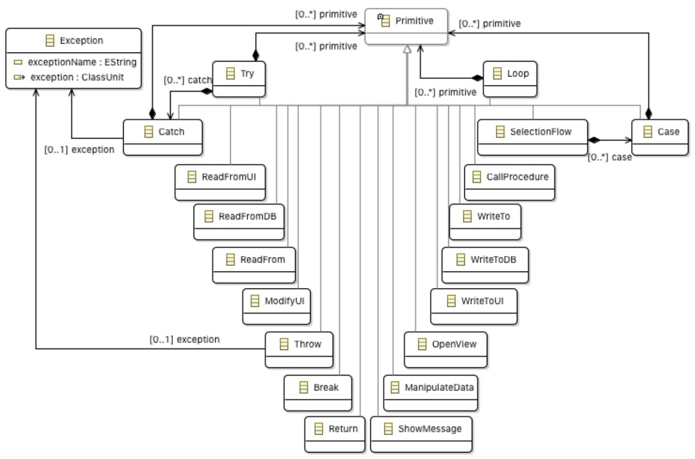


Figure 14: Primitive hierarchy in the Idioms metamodel.

For the sake of simplicity, the classes that represents expression elements have not been shown in the Figures 14 and 13, such as (i) operators (e.g. `And`, `Or`, and `Less`), (ii) the value of a variable (`VariableRef`), (iii) the return value of a function (`ReturnValue`), (iv) kinds of variables: UI widget variable (`UIVar`), local variable (`LocalVar`), global variable (`GlobalVar`), and (v) constant values (`Constant`).

Table 2: Some idioms defined for PL/SQL.

PL/SQL Code	Idiom/Meaning
Variable name	*ReadFrom* Read value from a local variable
Variable name := (Assignment)	*WriteTo* Write value to a local variable
UI Variable	*ReadFromUI* Read value from a UI variable
UI Variable := (Assignment)	*WriteToUI* Write value to a UI variable
Select	*ReadFromDB* Database operation that reads data
Insert/Update/Delete	*WriteToDB* Database operation that insert/update/delete data
*Operators (+, -, /, *)*	*ManipulateData* Arithmetical operation
Expressions (<, >, ...)	*Expression (Less, Greater, ...)* Boolean expression
Builtin function e.g. clear_item	*ModifyUI* Modify an UI property
IF / Switch	*SelectionFlow* Execution flow according to conditions
Then / When / Else / Elsif	*Case* Execution flow for an specific condition
While, For, Loop	*Loop* Repeatable code
Break	*Break* Ends a loop
Procedure/function call	*CallProcedure* Procedure call
Return	*Return* Returning value for a funtion/procedure
Try	*Try* Code block to execute
Catch	*Catch* Code block executed after an exception raised
Throw or Raise	*Throw* Raise an exception

7.2.2. *Implementation*

As indicated in Section 5, we have organized the model-to-model transformations in four components: *Iterator*, *Analyzer*, *Builder*, and *Reference Resolver*. In the case of the *kdm2idioms* transformation, these four components have been implemented to perform the following actions: (i) to iterate over the input KDM model, (ii) to analyze KDM model to discover the idioms, (iii) to create elements that form the Idiom model and build the aggregation hierarchies, and (iv) to establish references between the idioms newly created and the existing idioms.

For each element in the Idiom model, the *Iterator* invokes the *Analyzer* which is in charge of discovering idioms in the code. The *Analyzer* use the PL/SQL to Idioms mappings (see Table 2) to decompose code in terms of idioms. Once a new idiom is identified, the *Analyzer* invokes the *Builder* to create the corresponding model elements. The KDM elements from which a new idiom instance will be created are passed as argument in such an invocation because they provide the information needed to initialize the elements. Finally, the *Reference Resolver* connects the idiom created to one or more previously created idioms. It should be noted that idioms elements maintain a reference to the source KDM elements in order to keep a traceability to the code that forms a particular usage of a idiom.

7.2.3. *Testing*

The transformation has been implemented in an incremental way, as explained in Section 4. In this case, the process was organized in two phases. In the first phase, we injected 43 input KDM models for simple triggers that have only one sentence. With these triggers, we checked expressions, assignments, and statements (e.g. IF, CASE, and LOOP) which only includes code blocks formed for an assignment. In the second phase, we inject 24 KDM models for triggers containing nested statements that involves one or more idioms (e.g. IF or LOOP nested, and TRY-CATCH).

In both phases, for each KDM model injected, we first write the methods that implement the corresponding KDM-to-Idioms mapping. Then, the transformation is executed for the input model, and the output model is visually analyzed to validate it. For this checking, we have to navigate through the input KDM model and explore the trigger code. The tree editor provided by EMF is used to navigate through the models.

KDM models contain a large set of elements, even for simple code fragments. This in mainly due to the high number of *read* and *write* elements. For this reason, we decided to use the trigger code to validate the transformation, additionally to KDM models. Because the high level of abstraction of Idioms models, it is easy to check if a source code is correctly represented in terms of idioms.

7.2.4. *Application to the Trigger Example*

Figure 15 shown the Idioms model obtained for the example. A `Code` instance has been created which aggregates the six local variables of the trigger example and the four idioms identified: a `Try` (block from line 2 to line 16), two `SelectionFlow` (lines 18 and 25), and a `Write` (write to a variable in line 24). In turn, the `Try` element includes the following idioms: two `WriteTo` (write to a variable in lines 3 and 7), two `ReadFromDB` (SELECT operations in lines 4 and 5), a `SelectionFlow` (IF sentence in line 9), and a `Catch` that includes a `CallProcedure` (call in line 14) and a `Throw` (exception triggered in line 15). Idioms elements that are part of those mentioned above are not shown for the sake of simplicity.

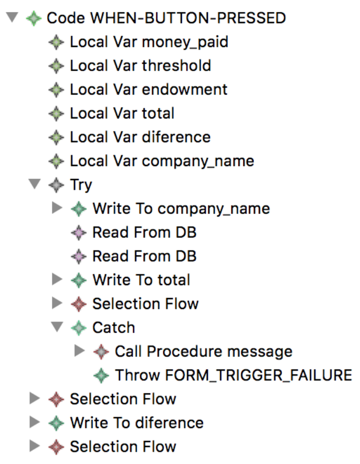

```
▼ Code WHEN-BUTTON-PRESSED
    Local Var money_paid
    Local Var threshold
    Local Var endowment
    Local Var total
    Local Var diference
    Local Var company_name
  ▼ Try
    ▶ Write To company_name
      Read From DB
      Read From DB
    ▶ Write To total
    ▶ Selection Flow
    ▼ Catch
      ▶ Call Procedure message
        Throw FORM_TRIGGER_FAILURE
  ▶ Selection Flow
  ▶ Write To diference
  ▶ Selection Flow
```

Figure 15: Idioms model example.

7.3. *Restructuring: Generation of the Target Platform model*

The monolithic code of PL/SQL trigger and program units has to be separated into the methods and classes that correspond to the tiers of the desired MVC architecture: JSF managed beans and Spring services in our case. This is achieved through the two model-to-model transformations that restructure the legacy code in several tiers. First, the *idioms2platform* transformation generates a *Target Platform* model that represent the classes and method in *View* and *Controller* tiers. The model obtained has references to `Method` objects that are part of a model that represent the object-oriented code to be generated. This model is generated in a second m2m transformation, named *platform2oo*, that takes as input the *Platform* model and the *Idiom* model. Therefore, this second transformation will first translate PL/SQL code into object-oriented code for each method in the *Platform* model.

We will here describe the first transformation, and the second one in the following section. We will begin by presenting the Target Platform metamodel that represents the MVC architecture defined in Section 6, more specifically the JSF and Spring elements involved in the implementation. It should be noted that although this transformation is platform-specific, the second one generates language-independent object-oriented code.

7.3.1. *Target Platform metamodel*

Figure 16 shows the Target Platform metamodel. `TargetPlatformModel` is the root class of the metamodel. A `TargetPlatformModel` aggregates three elements: a `Service`, a `UserInterfaceView`, and a `ManagedBean`. A `Service` represents a *Spring* controller composed of a set of

ServiceMethod methods that are invoked from event handlers or unit programs procedures. A UserInterfaceView represents an application window and it is made up of a set of UserInterfacesComponent (GUI widgets). A ManagedBean represents JSF managed bean. This class is the central element of this metamodel because it is connected to the rest of the elements. A ManagedBean aggregates (i) the set of event handlers (EventHandler) of the UserInterfaceView to which it is associated, and (ii) a set of attributes (ManagedBeanAttributes) that denote what UI fields the managed bean manipulates. Moreover, a ManagedBean references zero or more Services that includes the methods ServiceMethod that it calls. It is worth recalling that EventHandler references a Code element (i.e. the PL/SQL code of the event handler in form of Idioms) and a Method element (i.e. the Java code obtained when translating the PL/SQL code). Finally, two kinds of ServiceMethod have been defined: HelperServiceMethod for methods created for an operation enclosed in a program unit, and EventHandlerServiceMethod for methods created from the trigger code. Like an EventHandler, an EventHandlerServiceMethod contains references to Code and Method elements. Instead, a HelperServiceMethod only references a Method. The need of this distinction will be evidenced in explaining the implementation of the following two transformations.

7.3.2. Implementation

We have built the four components in which have organized our m2m transformations. When iterating the input Idiom model, it is required to also navigate over the KDM model because the Idiom elements do not contain references to the windows of the application. For this, we take advantage of the existing traceability from *Code* elements to the KDM elements that form it. A specific *KDMNavigator* class has been created to navigate over the KDM elements, which is used by the *Iterator* component.

For each Code element in the Idioms model, the CallableUnit KDM element that references it is accessed. From this element, a bottom-up navigation is performed from the GUI component to which the trigger is associated (UIResource KDM element) to the parent application window (Screen KDM element). Then, a UserInterfaceComponent is instantiated, which references to the UIResource that is source of the event handled by the trigger. When instantiating the first UserInterfaceComponent, a UserInterfaceView, ManagedBean, and Service object are then created. Each created UserInterfaceComponent is added to the UserInterfaceView and is referenced from the ManagedBean. Whenever a UserInterfaceComponent is instantiated, an EventHandler and a ManagedBeanAttribute are also created, which references it. The attribute code of EventHandler is initialized with a reference to the current Code. As indicated above, each EventHandler object has a reference to the Method object that represent its object-oriented code. These references will be void when the *idiomsp2Platform* transformation is completed, and they will be initialized during the execution of the *platform2oo* that has as input the Platform model.

When the visited Code element refers to a Program Unit, only a Service is created (not a ManagedBean). A Program Unit is identified if the CallableUnit accessed from the Code element does not contain a reference to an UIResource but it contains a CodeFragment stereotype.

Finally, it should be noted that ServiceMethods for a Service generated for a UserInterfaceComponent can not be created in this transformation, because they are created depending on the structure of the trigger code. The analysis of this structure is performed in the following transformation, when idioms are translated into object-oriented constructs. Instead, only ServiceMethods for a Service that are originated from a Program Unit are generated. This explains the existence of two kinds of ServiceMethods.

7.3.3. Testing

The transformation has been implemented incrementally in two phases. First, we have injected a KDM model for different program units in an input form. We checked that the target platform had a service and as many methods as program units. Second, we created more complex forms by adding different kinds of widgets, nesting windows, and having more than one window, in all the cases having a trigger associated to each visual component contained in the form. Then, we checked that (i) the platform model generated included a UserInterfaceView for each window and a UserInterfaceComponent for each widget, (ii) a ManagedBean and a Service for each window, and (iii) a ManagedBeanAttribute and EventHandler for each visual component.

7.3.4. Application to the Trigger Example

Figure 17 shows the main elements of the target platform model generated for the example: (i) one ManagedBean, (ii) one UserInterfaceView, and (iii) two Services. The ManagedBean contains one ManagedBeanAttribute that refers to the UserInterfaceComponent representing the button and an EventHandler that contains a reference to the event handler method. The GrantsService has not EventHandlerServiceMethods because them are generated in the following transformation, while the GrantsAppService service contains a HelperServiceMethod that corresponds to the function normalizeCompanyName that is part of the program unit. This service method has a null value for the attribute method as the Method instance will be created in the following transformation as explained above. Finally, the UserInterfaceView element aggregates an UserInterfaceComponent element which correspond to the only widget that it contains (i.e. a *button*). Note that only those visual components that have a *trigger* associated are aggregated.

7.4. Restructuring: Generation of the Object model

We shall describe the *platform2oo* transformation that, taking the Idioms and target platform models as input, is able to

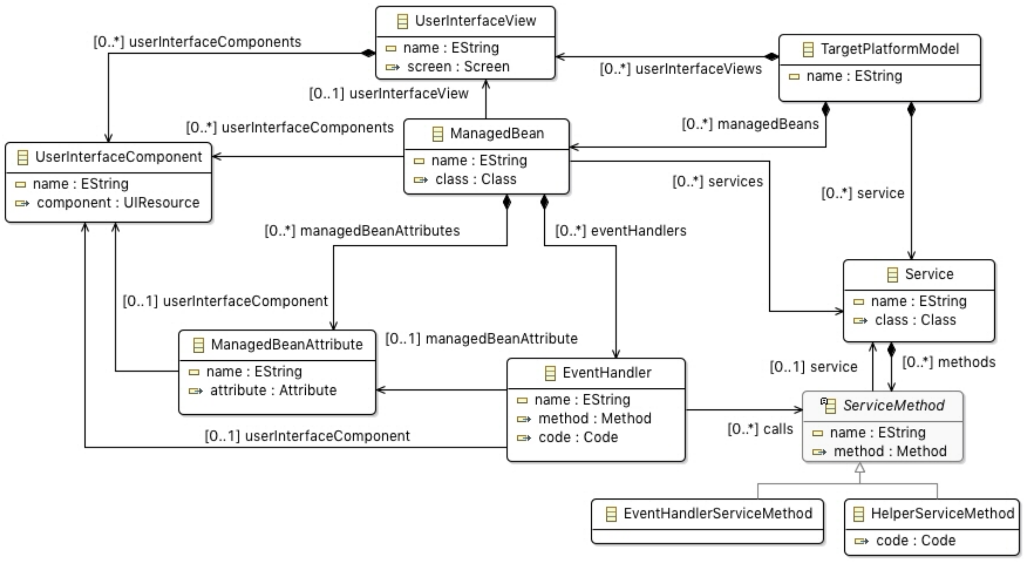


Figure 16: Target Platform Metamodel.

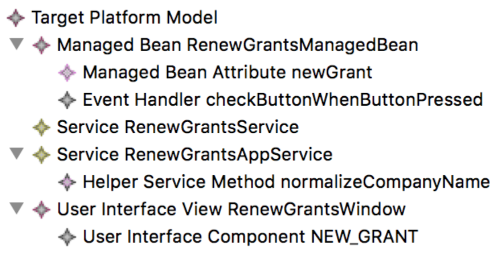

Target Platform Model
Managed Bean RenewGrantsManagedBean
Managed Bean Attribute newGrant
Event Handler checkButtonWhenButtonPressed
Service RenewGrantsService
Service RenewGrantsAppService
Helper Service Method normalizeCompanyName
User Interface View RenewGrantsWindow
User Interface Component NEW_GRANT

Figure 17: Target Platform Model example.

generate the Object-Oriented model. While the target platform model specifies which classes and methods must be generated for each tier of the MVC architecture, the Idioms model provides an abstract representation of the legacy code. Therefore, the target Platform model directs the transformation, and the Idioms model is needed to generate the object-oriented representation of the code to be included in each method. First of all, we will present the Object-Oriented metamodel.

7.4.1. Object-Oriented metamodel

Figure 18 shows the Object-Oriented metamodel. To define this metamodel, we reused the Java metamodel provided by the Modisco project [29], modified in order to convert it to a language-independent metamodel. Next, we will comment the main elements of this metamodel.

An *Object-Oriented model* consists of a set of `Types` and `Moduless`. A `Class` contains a set of `Attributes` and `Methods`. A `Method` is composed of a header (`MethodDeclaration`), a body (`MethodBody`), and local variables (`Variable`). The statements of a method body are represented as a set of `Statements`. `Statement` is the root of a hierarchy representing the kinds of sentences. `StatementContainer` inherits from `Statement` and, in turn, is the root of statements that can include a code block as: conditional (`If` and `Switch`) loops (`For`, `While`, and `ForEach`), and exception handlers (`Try` and `Catch`). Other kinds of statements are expressions (`Expression`), assignments (`VariableAssign`) method calls (`MethodCall`), and variable declarations (`VariableDeclaration`).

7.4.2. Implementation

The Iterator component traverses the `ManagedBean` and `Service` elements contained in the Platform model. For each `ManagedBean` and `Service`, the Builder component creates a `Class` whose name is formed by a window identifier followed by "ManagedBean" or "Service", respectively. The window identifier results of concatenating the window name with the value of a counter used to accumulate the number of windows found in a form. Once these classes are created, the transformation must generate `Methods` and `Attributes` for each of these classes. This generation is done differently for `ManagedBean` and `Service` as explained below. Note that the process performed involved the components Analyzer, Builder and Reference Resolver.

Services processing. As explained above, the Platform model contains a `Service` that aggregates a `HelperServiceMethod` for each operation that is part of a program unit. For these services, a `Method` is generated for each `HelperServiceMethod`. The `code` attribute of a service method is used to access the set of idioms that represent the corresponding program unit operation. These idioms are translated into the object-oriented statements that form the created method body. In this way, we have a `Service` class with a method for each operation included in a program unit.

ManagedBean processing. Algorithm 1 depicts the processing of the event handlers of a managed bean. The set of `EventHandlers` that aggregates a `ManagedBean` are traversed in order to apply the code processing (line 1) on each of them.

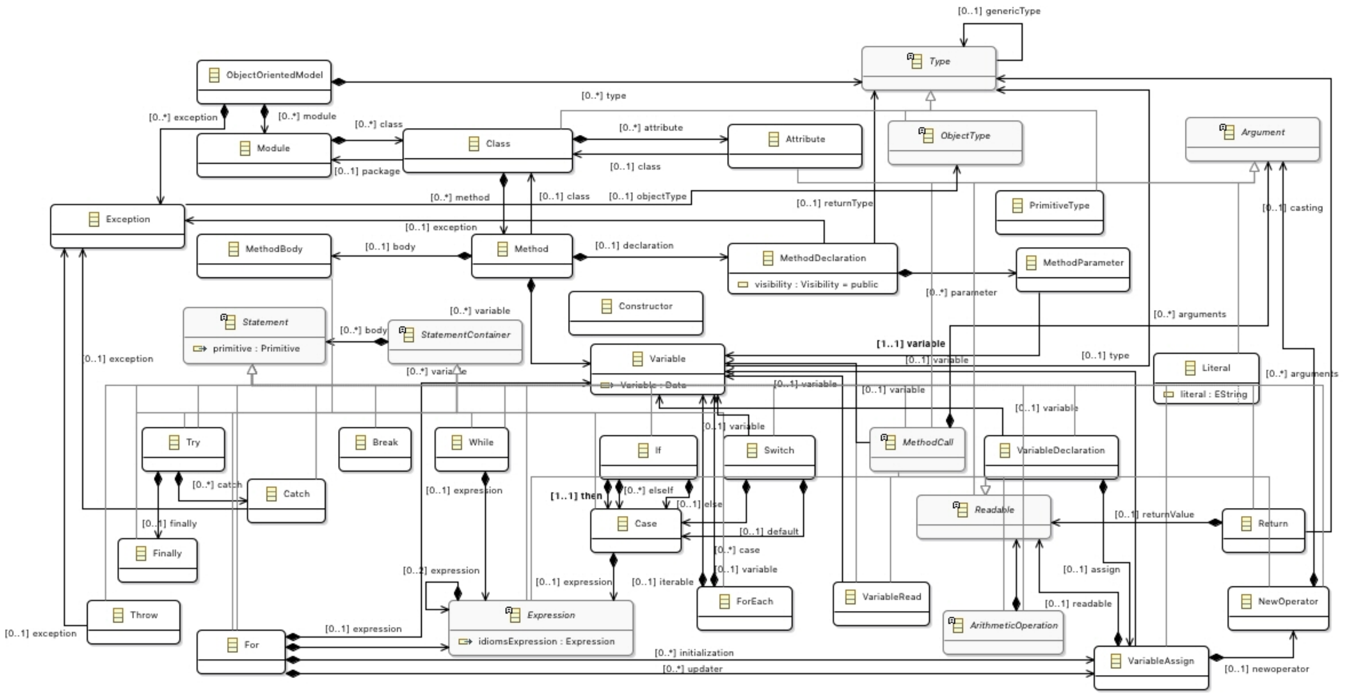


Figure 18: An Object-Oriented metamodel.

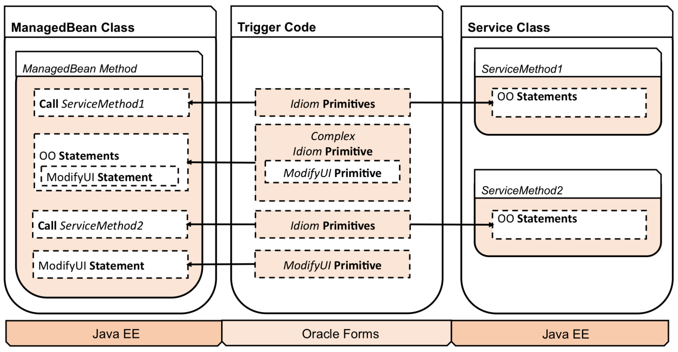


Figure 19: Separating trigger code into managed bean and services.

First of all, a new `Method` is created containing the statements of the event handler code in the managed bean (line 6). Given the set of idioms describing a trigger (the `Code` instance referenced by the `code` attribute of the event handler) and the created method (the `method` attribute), the algorithm first creates another method that is added to the service class (line 10), and a method call that is added to the managed bean method (line 11). When the service method is created, a `EventHandlerServiceMethod` is also created and aggregated to the `Service` instance of the Platform model. Then, the idioms that form an event handler are traversed in order to separate code manipulating view elements from the rest of the code (line 13). Figure 19 graphically illustrates how this code separation is done. Each idiom is processed to check whether it is a `ModifyUI` idiom (line 27). This processing first translates each idiom into a set of object-oriented statements, which are added to a list of statements (line 29), and after it is checked if the current idiom is a `ModifyUI` (line 35) or either it has other nested idioms. When nested idioms are found, they are recursively processed (line 33). Whenever a `ModifyUI` is found four actions are sequentially executed: the last idioms processed are translated into OO statements and moved to the current service method (line 15), a new service method is created (line 16), a method call to this method is created and added to the managed bean method (line 17), and shared variables are also moved to the new service method (line 18).

The problem of identifying what variables are being shared among methods that result of code separation is not trivial. The previous algorithm also declares and shares variables by means of storing data about accesses to variables. These information allow us to discover those variables which are being used in several methods and therefore which of them are being shared. When the algorithm detects one read/write (in the `processPrimitive` function), the primitive is processed as a local variable access and all the information is stored: the variable, the variable access, and the method where the variable is being used. The method can be a service or a managed bean. Special consideration must be taken into account when the variable access is done inside a complex primitive that contains a `ModifyUI` idiom. Then, the code of the complex primitive is moved from the service to the managed bean. In addition, the method associated to the variable access is updated. Once all the idioms of the code trigger have been processed, the variable accesses contained in a method are compared to all the variable accesses of the rest of methods. When a variable access is present in two or more methods then it is required to change the local accesses of the variable by the use of a map. The map is in charge of sharing variable values between different methods by sharing the same map instance among all of them.

When the variable is only accessed by one method, then a local variable declaration inside the method is produced. Because in PL/SQL code variables are declared at the beginning of the trigger procedure, the algorithm requires to calculate the adequate place where to introduce the variable declaration.

In the following, we describe a new part of the algorithm for discovering the code blocks where a variable declaration must be introduced (see Algorithm 2). The first variable access is set as the variable declaration. Then, the algorithm iterates over all the variable accesses, and it compares each one to the initial variable declaration to check in what code block the variable declaration must be implemented. For each pair of initial variable declaration and variable access, the block where both are included is compared. If both blocks are to the same, the first variable access is chosen as the variable declaration (line 4). For instance, a source code as `age := 18` will produce the next target code: `int age = 18;`. If blocks are different, then it is checked if one block is containing the other one. In that case, the outer block will contain the variable declaration (lines 6 and 8). If blocks are different and there is not a containment relation, then the first block containing both blocks is chosen to implement the variable declaration (line 10). For example, the next source code `if (age >= 18) adult = true; else adult = false` will determine that declaration must be in the block immediately containing the block `if (boolean) ... else ...`. Finally, if the first variable access corresponds to a read, a code comment is introduced in order to inform that a variable is being used and could not have been initialized. It is worth recalling that the Java compiler initializes to `0` all `int` variables not initialized explicitly in the code. However, in PL/SQL the meaning of an initialization absence would be the assignment of a *NULL* as initial value, whichever the variable type was.

7.4.3. Testing

The part of the transformation that concerns to the procedural-to-object translation has been incrementally built by reusing the idioms models that were generated during the implementation of the *kdm2idioms* transformation. Recall that we injected 67 KDM models which covered the set of defined idioms. Therefore, we have checked that the Object-oriented model generated for each of the idioms model is correct. For this, we have manually explored each Idiom model and checked that the expected object-oriented statements have been generated for each idiom. Each of the object-oriented models has been easily validated, because the mapping between idioms and object-oriented elements is simple.

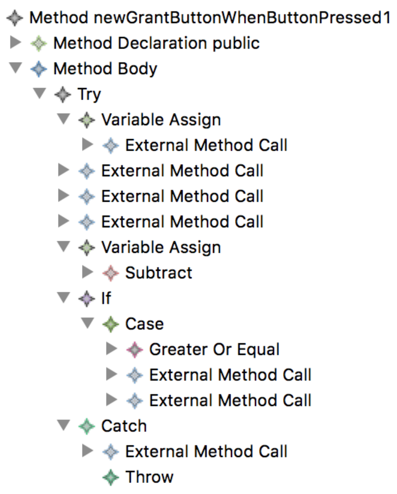
Regarding the implementation of the separation of code, the transformation has been validated through the code generated by the *objectual2java* transformation. The direct correspondence between the object-oriented model and the generated Java code justifies this decision. We found that the manual validation of the code separation was easier on Java code editors than using modelling editors. In addition, the Java compiler

Algorithm 1 Separation of Managed bean and Service code.

```
for $eventHandler \in managedBean.eventHandlers$ do
    $processCode(eventHandler)$
end for

procedure $processCode(eventHandler)$
    $eventHandler.method \leftarrow createMethod()$
    $method \leftarrow eventHandler.method$
    $code \leftarrow eventHandler.code$

    $serviceMethod \leftarrow createServiceMethod()$
    $method.createServiceCall(serviceMethod)$
    for $idiom \in code.primitives$ do
        $moveToUi \leftarrow processIdiom(idiom, serviceMethod.statements)$
        if $moveToUi$ then
            $serviceMethod.moveLastStatementToMethod(method)$
            $serviceMethod \leftarrow createServiceMethod()$
            $method.createServiceCall(serviceMethod)$
            $method.moveVariablesToMethod(serviceMethod)$
        end if
    end for
    if $service.statement.isEmpty()$ then
        $method.removeLastServiceCall()$
        $deleteserviceMethod$
    end if
end procedure

function $processIdiom(idiom, statements)$
    $OOstatement \leftarrow doMapping(idiom)$
    $statements.add(OOstatement)$
    $moveUI \leftarrow false$
    if $isComplexIdiom(idiom)$ then
        for $nestedIdiom \in idiom.primitives$ do
            $moveToUI \vee processIdiom(nestedidiom, OOstatement.statements)$
        end for
    else if $isModifyUIIdiom(idiom)$ then
        $moveUI \leftarrow true$
    end if
    return $moveUI$
end function
```

Algorithm 2 Declaring variables in the right code block.

```
1:  $variableDeclaration \leftarrow firstVariableAccess$
2:  for $access \in restOfVariableAccesses$ do
3:      $declarationBlock \leftarrow empty$
4:      if $variableDeclaration.block == access.block$ then
5:          $declarationBlock \leftarrow variableDeclaration.block$
6:      else if $variableDeclaration.block \neq access.block \wedge$
    $access.block \in variableDeclaration.block$ then
7:          $declarationBlock \leftarrow variableDeclaration.block$
8:      else if $variableDeclaration.block! = access.block \wedge$
    $variableDeclaration.block \in access.block$ then
9:          $declarationBlock \leftarrow access.block$
10:     else
11:         $declarationBlock \leftarrow allBlocks.selectFirst(block|$
    $block.contains(variableDeclaration.block) \wedge$
    $block.contains(access.block))$
12:     end if
13:     if $isOnlyRead(firstVariableAccess)$ then
14:         $declarationBlock.addComment("//Variablenotexplicitlyinitialized")$
15:     end if
16: end for
```



```
Method newGrantButtonWhenButtonPressed1
  Method Declaration public
  Method Body
    Try
      Variable Assign
        External Method Call
      External Method Call
      External Method Call
      External Method Call
      Variable Assign
        Subtract
      If
        Case
          Greater Or Equal
          External Method Call
          External Method Call
      Catch
        External Method Call
        Throw
```

Figure 20: An excerpt of the Object-Oriented model for the example.

checks automatically if a variable was declared in the right code block.

7.4.4. Application to the Trigger Example

Figure 20 shows an excerpt of the Object-Oriented model obtained for the example. Specifically, the method `newGrantButtonWhenButtonPressed1` included in the class `RenewGrantsService`. The method contains the elements that corresponds to the code shown in Section 6. It starts with a `Try` and several calls to methods (in this case, first the call to `normalizeCompanyName`, then the different `readFromDB`). Then, after the assignment of the variable, the `If`, and finally the corresponding `Catch`, with its body.

7.5. Forward engineering: Code generation

This Section describes the model-to-text transformation named *objectual2java* that generates the final application code from the Objectual model. This transformation has been implemented in Acceleo [30]. Acceleo is an implementation of the Mof2Text standard proposed by OMG to write M2T transformations [31].

The transformation consists of three modules: `ClassTemplate`, `ClassParts`, and `Statements`. `ClassTemplate` only contains a template that generates the structure of a class by invoking the templates defined in `Statements`, which generates import declarations, attributes declarations, and method declarations. The statements form the body of the methods. The `Statements` module has a template for each kind of statement, and all these templates are overloaded, that is, they have the same name and parameter type. Because all the statements are instances of classes inheriting from the `Statement` class, a precondition can be established in each template to indicate the kind of statement for which is applied.

This transformation had to be integrated with the *UI2Java* m2t transformation created by Open Canarias as part of the GUI Migration tool in order to generate code for the user interfaces. `ManagedBean` class skeletons are generated by the *UI2Java* transformation. Therefore, our *objectual2java* transformation must generate all the code for service classes and code for the method body of the `ManagedBean` classes. The strategy agreed with the company to integrate both transformations is the following: *UI2Java* templates invoke to objectual2java templates passing a `CallableUnit` KDM element as argument. Then, we traverse the Platform model to find the `EventHandler` whose `Code` references this `CallableUnit`, and we generate Java code for the `Method` element referenced from that `EventHandler`. Therefore, the *objectual2java* transformation has the *Object-Oriented* and *Platform* models as input.

7.5.1. Testing

Because the object-oriented models are representations that are very close to the Java code, the Acceleo templates are simple and they are therefore easy to test. They have been validated as follows. First, the `ClassTemplate` and `ClassParts` modules have been created. Then, the `Statement` module has been incrementally developed. We have followed the same strategy used to validate Object-oriented models. That is, we have used the models object-oriented generated from the 67 KDM models injected. In fact, object-oriented models and Java code are validated at the same time. Therefore, this model-to-text transformation is written the same as the *platform2objectual* transformation is written. Generated code was compiled for syntactic and semantic validation. Moreover, a manual validation was carried out to check if indentation was correct for each line of code, and statements follow the correct order.

Once the *objectual2java* transformation has been completed, we checked that the `ManagedBean` and `Service` classes have been correctly generated, which implies to perform five kinds of checks: (i) a managed bean class and a service class has been generated for each trigger in a form; (ii) a service class has been created for all the program units of a form, and (iii) the methods of each class have been generated, and they have the expected code. This validation has been carried out for the three forms presented in the following section.

7.5.2. Application to the Trigger Example

The code generated for the trigger example is shown in Section 6. As can be observed there, the following artifacts were generated:

- A managed bean class `RenewGrantsManagedBean` with a event handler method `newGrantButtonWhenButtonPressed`.
- A service class `RenewGrantsService` with the two methods that are invoked from the event handler method: `newGrantButtonWhenButtonPressed1` and `newGrantButtonWhenButtonPressed2`. The first one corresponds to the excerpt of Object-oriented model shown in Figure 20.
- A service class `RenewGrantsAppService` for the program unit included in the form, which contains the method `normalizeCompanyName`.

It is worth noting that the service class `RenewGrantsService` has two methods because the trigger example has only a modify idiom whose code is shown below.

```
IF money_paid >= threshold THEN
   SET_ITEM_PROPERTY('RENEW_COMPANY_GRANTS.
        GRANT_RENEWED', visible, property_true);
ELSE
   SET_ITEM_PROPERTY('RENEW_COMPANY_GRANTS.
        THRESHOLD_NOT_EXCEEDED', visible,
        property_true);
END IF;
```

The Java code generated for this `Modify` idiom would be:

```
if ((Double)map.get("moneyPaid") >= (Double)map.get
      ("threshold")) {
   setRenewCompanyGrantsGrantRenewedVisible(true);
} else {
   setRenewCompanyGrantsThresholdNotExceededVisible(
        true);
}
```

8. Validation of the tool

This Section is devoted to describe the validation of the final code generated by the tool (i.e. the output of the transformation chain that implements the reengineering process). We first establish the scope of the validation by providing a definition of all those testings involved in the validation. Then, a description of the instrumentation and methodology applied on is given. Finally, we present and comment the result of the validation. In addition, the threats to validity that should be considered in order to accept the result, are identified.

8.1. Definition

Validating a model transformation chain entails to test each transformation as well as to check whether the final software artifacts generated meet the initial requirements. In the case of a code migration, these requirements establish how the legacy code must be translated into target code.

In the previous Section, we have explained the black-box testing approach applied to validate each of the transformations in the chain. Here, we will explain the tests performed to validate that the tool produces the expected code. As indicated in Section 5, three kinds of black-box tests have been considered to verify how accurate the obtained results are, and how they conform to source-target mapping.

- *Unit tests for generated Java code*. Unit tests have been written and executed for all the generated classes.
- *Coverage tests*. Some software metrics have been used to measure the migration coverage. Coverage tests are executed to know how much code constructs in the legacy application have been migrated. These measurements are useful to verify that all the legacy code has been processed during the migration process.
- *Acceptance tests*. Functional requirements have been finally validated by means of acceptance tests. For this, the company has checked that legacy PL/SQL code and generated Java code have the same behavior for the three forms described later.

8.2. Instrumentation

JUnit has been used to write and run the unit tests for the generated Managed Bean and service classes. Additionally, we have used a Java compiler for checking the syntax and some semantic aspects as method invocations.

The coverage testing has involved the use of tools for software inspection. Concretely, we have used the ClearSQL7 and Eclipse Metrics 3 tools for obtaining metrics from Oracle Forms applications and Java code, respectively. The metrics measure the number of triggers, program units, and SQL sentences for the source application, and the number of methods and SQL sentences for the target application. It is worth recalling that triggers and program units are migrated to methods in the target platform. In order to validate the intermediate models, we have implemented short Java/EMF programs for counting elements in the Ecore models.

Acceptance tests have been manually performed by testers from the company. They have executed the legacy and target applications to check whether UI event handlers have the same behavior.

The three kinds of testing have been carried on three legacy forms provided by the company, which have different size: (i) a *small* size form containing 55 triggers and 25 program units; (ii) a *medium* size form with a total of 44 triggers and 39 program units; and (iii) a *large* size form with 260 triggers and 71 program units.

8.3. Methodology

For each metamodel, we have built a short Java program whose purpose is to calculate the metrics defined for coverage testing. At the end of the implementation of each model transformation, we measure the metrics by executing the Java programs that corresponds to the target metamodel. In addition, we have also manually calculated the metrics by inspecting the target model with the EMF tree editor. In the case of the Idiom model, we make sure that the model elements are also present in the KDM model and the source code. Platform models have been validated by checking if each UI widget, event handler, window, trigger and program unit has been correctly injected. Object-oriented models are validated in a similar way as Idioms models. Moreover, these models are indirectly validated through testing on the generated code, because there are a direct correspondence between them.

Once the model transformation chain was developed, unit tests were first tackled. Before writing the unit tests, the generated code was modified to include some mocks to simulate the built-in functions and the data access methods invoked. Then, unit tests were written for the managed bean and the service classes. The unit tests were performed following a bottom-up strategy according to the dependencies among classes. First, the unit tests for the unit programs service class were performed, then for the service class implementing the data access and the application logic, and finally for the managed bean class that includes the event handler methods that invoke the methods of the service classes.

After running the unit tests, we measured the metrics that we mentioned before. The results are shown in Table 3, and are commented below.

We performed integration tests on the services, without considering the calling methods to mocks. Then we proceed in a similar way on the managed beans.

Finally, company testers executed the generated code interacting with the generated JSF views for the three tested forms. As explained in Section 7.5, the methods generated by the *Objectual2Java* transformation, are invoked from the templates generated by the UI Migration tool. Therefore, the managed bean and service classes are automatically integrated in the views classes generated by the UI Migration tool. Prior to perform the acceptance tests, company members developed the integration tests for the data access services to validate the integration of the managed bean classes with the service classes. As a result of the acceptance tests, some errors were reported, for example, identifiers that were declared as local variable and parameter in the same method, missing import sentences, the `i++` sentence was generated as `i=+`, or some primitive type variables with a NULL value. These errors were fixed in the model transformation chain.

8.4. Results

The results obtained for the three tested forms are shown in Table 3. The column *#elements* indicates the measured metrics. As depicted in the table, the metrics count the number of legacy artifacts of a certain type and the artifacts generated in their migration. These artifacts are the following: (i) Forms Triggers and methods in the generated managed bean class; (ii) Program Units and methods in the service classes that where generated to implement program units, and finally (iii) SQL sentences, that are present in both source and target code. The second column of the table indicates the size of the tested form from which the metrics were obtained: small, medium, and large. The rest of columns indicate the value of the metrics of the first column for a particular artifact involved in the transformation chain: (column 3) Oracle Forms; (column 4) KDM and Idioms models; (column 5) Platform and Objectual models; and (column 6) Java application.

We have used ClearSQL for counting the triggers, program units, and SQL sentences. We have also used this tool to find which triggers are empty. Then we have subtracted these empty triggers from the total number of triggers. Note that the existence of empty triggers is common in Oracle Forms applications (mostly derived from human errors). The Metrics tool has been used for counting the number of methods in Java classes.

With regard to the methods generated from the triggers migration, it is worth noting that there are less methods than triggers. We have detected that some triggers were missing during the injection because they were not associated to an UI widget but they correspond to data block triggers. Concretely, 1, 12, and 5 triggers were not generated for the small, medium and large forms, respectively. In the case of program units, no methods were missed.

Table 3: Results

#elements		**Forms**	**KDM/Idioms**	**Platform/OO**	**Java**
Triggers / Methods					
	small	5	5	4	4
	medium	42	42	30	30
	large	251	251	246	246
Prog. Units / Methods					
	small	25	25	25	25
	medium	30	30	30	30
	large	68	68	68	68
SQL sent.					
	small	13	13	13	13
	medium	59	59	70	70
	large	169	169	200	200

The number of SQL sentences in Java code (and Object-oriented models) is bigger than in Forms code and KDM/Idioms models. This is due to the strategy adopted for migrating some SQL sentences. When a query sentence includes a `SELECT ... INTO` structure for assigning two or more columns into two or more variables, our solution splits the original sentence in two or more new sentences where only one column is projected into one variable.

8.5. Limitations of the validation

In this section we will enumerate some limitations of the carried out validation.

1. There are no unit tests for some of the generated Java code (some methods have not been tested) but coverage is of 46% (222 LOC) for the small size form, 49% (687 LOC) for the medium size form, and 23% (1065 LOC) for the large size form.
2. The three validated forms migth not cover all the elements of the PL/SQL grammar or all the potential artifacts in the Oracle Forms architecture.
3. Potential mistakes when contrasting behavior of the generated code against the requirements of the legacy application.
4. Potential mistakes in counting elements when a visual inspection is used for the EMF tree editor.
5. Potential errors can be found when counting triggers in Oracle Forms by using ClearSQL7 because an empty trigger is counted by the tool but not considered in our solution.

9. Evaluation

Metamodels and model transformations are the two essential elements in developing a MDE solution. In this Section, we will analyze the main issues related to them in the context of our migration work. Regarding the metamodels used, we focus on the KDM metamodel. On the other hand, we will discuss how model transformations have been written and tested. In this discussion, we will address issues as the visualization of the output models, and the feasibility of applying a test-first programming approach. Additionally, we will present a comparison carried between Java, ATL, and QVTo as m2m transformation languages. Finally, we will comment two issues of great interest in migration scenarios: how the horseshoe model has been implemented and how the model traceability has been managed.

9.1. Using KDM

When OMG launched KDM 1.0 in 2007, the company Open Canarias decided to adopt this metamodel, as it provided the level of abstraction appropriate to represent source code of legacy applications, as an alternative to use AST models or other existing metamodels, such as EGL proposed by IBM. Since then, this company has used KDM to represent GPL code in all its model-driven modernization projects. KDM has provided two main benefits to this company: (i) flexibility to model statements written in an ambiguous way, (ii) ability to reuse visualization and code analysis tooling for different GPLs. It is worth noting that Open Canarias contributed to the implementation of the ASTM metamodel in Modisco [32]. This company has developed infrastructure to create KDM injectors for GPLs, as explained in Section 7.1. This infrastructure has been applied to create injectors for COBOL, ABAP, and PL/SQL.

Below, we detail the knowledge gained when using KDM to develop the tool presented in this paper.

KDM compliance.
The cost of creating the PL/SQL injector was 3 months/man. The injector is L1 compliant KDM for Data domain but not for Micro-KDM. A family of stereotypes was defined to represent PL/SQL statements instead of using Micro-KDM, as explained in Section 7.1. This has not negatively influenced the migration implementation. Whenever an `ActionElement` is processed, the `name` attribute must be used to check if it represents a sentence of the expected type. KDM models could then be used as input of a tool [33] developed to implement the AFP specification [17]. This tool was developed during the migration project to calculate the function points of Oracle Forms applications, but could be applied to other languages.

KDM limitations to manage Data from Code.
According to the KDM specification, the elements of a package can have references to elements in packages of a lower level but not vice versa, as the models are constructed from the code to the more abstract viewpoints. This lack of navigability can make writing model transformation difficult. The injector used created references from `Action` elements to `UI` elements, but not to `Data` elements. This caused problems in writing model transformations that required access to the database schema. We solved this issue by directly moving the PL/SQL sentences to the Java code. References to `UI` elements are really needed as observed in the *Idioms2Platform* transformation. They allowed us to access to `UI` elements from `CallableUnit` in order to know to which visual component an idiom is associated, as described in Section 7.3.1.

Benefits and limitations of KDM to model software concerns.
Infrastructure and *Program Elements* layers are very useful to represent legacy code. L0 compliant KDM models are more convenient than AST or CST models to perform a reverse engineering process, and L1 compliant Micro-KDM models promote the interoperability. We have been able to experiment with these benefits in the work here presented. Data managed by legacy applications can also be adequately represented by using the `Data` package. However, we have not used this package in our work, as explained above.

From our experience, the rest of level 1 KDM packages are too generic to be useful as-is. The `UI` package is very limited to represent the UI information required in software modernization tasks. For example, this package only includes the generic `UIField` and `UILayout` classes to represent information on widgets and layouts, respectively. KDM should therefore be extended in order to represent different kinds of widgets and layouts. However, the KDM extension mechanism is very limited in practice as described in Section 2, and when using stereotypes the interoperability among tools (one of the main benefits of KDM) is lost.

Therefore, our understanding is that defining (or reusing a existing metamodel) tailored to the domain of interest is a better alternative to KDM extensions as also noted in [34]. In fact, Open Canarias decided to use IFML and UsiXML metamodels to represent the legacy UI, as indicated above. In short, KDM models are an appropriate representation to start a model-driven reverse engineering process, but other metamodels must be defined (or reused) to abstract the information managed throughout a re-enginering process. In the project described here, we have devised the Idioms, Platform, and Object-Oriented metamodels. It is worth noting that we have not used the metamodel defined in [8] to represent the execution flow of idioms since this flow is represented in KDM models.

The size of the KDM models for the large case study is nearly 60Mb, while the sum of the sizes of the other three models is 5Mb. This evidences that KDM models compliant L0 or L1 level represent code at a very low level of abstraction.

Learning of KDM.
Two of the members of our team learned KDM, in particular the `Infrastructure` and `Program Elements` layers and the `Data` and `UI` packages. They had a solid background in MDE and were able to acquire required KDM knowledge in 50 hours. This number of hours includes the time devoted to prepare a report about the injection process developed by Open Canarias. Writing this report required understanding on how each PL/SQL statement was represented by means of stereotypes. A tutorial previously elaborated by our group facilitated the learning of KDM. It is worth noting the lack of publicly available KDM tutorials.

9.2. Writing model-to-model transformations in Java

Software migration is a MDE usage scenario which involves complex model-to-model transformations. In Morpheus, Java (and EMF API) was considered a better option than using model-to-model transformation languages such as ATL [13], ETL [35], or QVT operational [14]. This choice was motivated by two main reasons: (i) immaturity and lack of stability of tooling existing for most popular m2m transformation languages, and (ii) the high complexity of the transformations involved would demand to write a large amount of imperative code. In fact, we had already used Java instead of ATL or RubyTL [36] in some recent reverse engineering projects [34]. Moreover, Open Canarias had performed an internal survey on m2m transformation languages, and the staff concluded the convenience of using GPL or domain specific languages embedded into GPL.

Once the construction of the Code Migrator tool finished, we conducted a study to compare Java with two common m2m transformation languages: ATL as hybrid language, and QVT operational as imperative language. We used one of the m2m transformations implemented as a case study. Since 2007, the *Model Transformation Contest* is held to compare m2m tools through a case study raised in advance. This workshop has originated the publication of some valuable comparative studies of m2m languages [37, 38]. Our comparison contrasts with these studies as follows: (i) the case studies are more complex and are taken from a real project; (ii) KDM is the source metamodel; and (iii) the same developer has written the transformations.

The three m2m transformations implemented have the following size measured in lines of code (LOC): 1926 for *kdm2dioms*, 535 for *idioms2platform*, and 4568 for *platform2oo*. We therefore chose the *kdm2idioms* transformation as case study because had a medium size and complexity. Writing the ATL and QVTo transformations, we considered the features of each language. ATL is a hybrid language that allows to express the mapping between the source and target metamodels in a declarative way, and provides imperative constructs to express more complex parts of a transformation. QVT Operational is an imperative language that is part of the QVT hybrid specification [14], but that can be used separately from the QVT relational language. Both ATL and QVTo provide rules and helpers to express transformations. Helpers are used to factorize code and achieve short and legible rules. The ATL rules have a special clause to write imperative code that is executed after applying mappings. Whereas ATL helpers can only include OCL expressions, QVTo helpers can include any kind of statement and the state of the transformation can be changed. In addition, QVTo allows to declare intermediate data by means of `attributes`. In ATL, the order in which rules are executed is implicitly determined, whereas this order must be explicitly expressed in QVTo code. Regarding the creation of target elements, this is implicit in ATL but explicit in QVTo.

As explained in Section 5, we have organized the model-to-model transformations in four components: Iterator, Analyzer, Builder, and Reference Resolver. How these components have been implemented for the *kdm2idioms* transformation was described in detail in Section 7.2.

The ATL and QVTo transformations have been organized as follows. In both cases, a rule has been defined for each mapping kdm-idioms. Intermediate data are required to record symbol tables for local and global variables and exceptions. These tables have been declared as properties (`dictionary` type), whereas the dynamic map pattern[5] has been applied in ATL. In ATL, we have encountered difficulties in writing the filters needed to discriminate the kind of statement represented by an `ActionElement` source element, and sometimes we had to explicitly invoke rules. Using QVT properties to store intermediate data instead of query helpers allowed us to reduce the execution time.

In ATL, the transformation has 37 declarative rules (444 LOC), 5 rules embedding imperative code (85 LOC), and 14 helpers (50 LOC). In QVTo, the transformation has 41 rules (316 LOC), 11 helpers (76 LOC), and 8 properties. The Java transformation has 1926 LOC distributed among the four components as follows: Iterator (325 LOC), Analyzer (148 LOC), Builder (1077 LOC), and Reference Resolver (376 LOC). It is worth noting that streams and lambda expressions in Java 8 provide an expressiveness similar to the OCL language to navigate models. Therefore, the effort devoted to write model navigation expressions could be considered equivalent in the three languages. Table 21 shows the code to count the number of exceptions in a KDM model for OCL, Java 8, QVTo and ATL. However, the creation of target elements is tedious in Java because factory classes and getter/setter methods must be used.

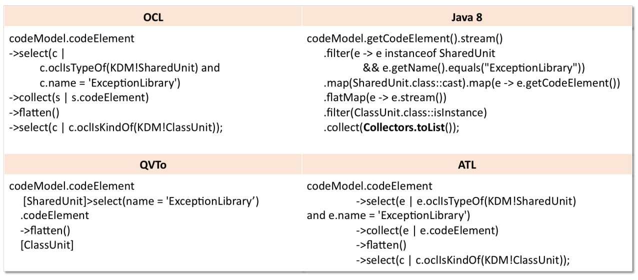

OCL
```
codeModel.codeElement
->select(c |
        c.oclIsTypeOf(KDM!SharedUnit) and
        c.name = 'ExceptionLibrary')
->collect(s | s.codeElement)
->flatten()
->select(c | c.oclIsKindOf(KDM!ClassUnit));
```
Java 8
```
codeModel.getCodeElement().stream()
    .filter(e -> e instanceof SharedUnit
            && e.getName().equals("ExceptionLibrary"))
    .map(SharedUnit.class::cast).map(e -> e.getCodeElement())
    .flatMap(e -> e.stream())
    .filter(ClassUnit.class::isInstance)
    .collect(Collectors.toList());
```
QVTo
```
codeModel.codeElement
   [SharedUnit]>select(name = 'ExceptionLibrary’)
  .codeElement
  ->flatten()
  [ClassUnit]
```
ATL
```
codeModel.codeElement
            ->select(e | e.oclIsTypeOf(KDM!SharedUnit)
and e.name = 'ExceptionLibrary')
            ->collect(e | e.codeElement)
            ->flatten()
            ->select(c | c.oclIsKindOf(KDM!ClassUnit));
```

Figure 21: Code comparison example of OCL, Java 8, QVTo and ATL.

The size and development time for each transformation has been the following: 579 LOC and 88 hours for ATL, 392 LOC and 48 hours for QVTo, and 1926 LOC and 160 hours for Java. It should be noted that writing the Java transformation required understanding the problem and designing the solution, which approximately involved half of the time spent in the implementation.

We have measured the performance of each transformation. Each transformation has been executed five times for two different inputs: a medium (87 elements) and large (190 elements) KDM model. The transformations have been executed under a Core i5 CPU at 2.7Ghz, 8Gb of RAM and 3Mb of cache, a SSD hard disk, MacOS Sierra 10.13.3, ATL 3.8, QVTo Eclipse 3.7 and Java 8. For each run, the execution time is calculated as the sum of the completion times for three tasks: load the input model, execute the transformation, and write the target model.

Figure 22 shows the average execution times for each input to the transformations. The variance obtained has been low in all the cases with values in the range between 0.01 and 0.06.

[5]Unofficial ATL Tutorial: https://github.com/jesusc/atl-tutorial.

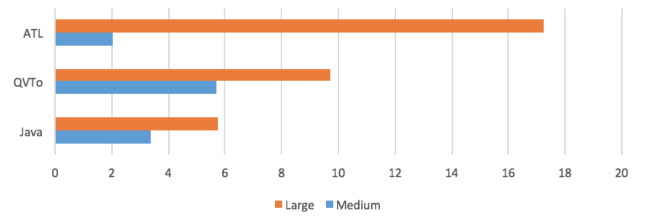


Figure 22: Average execution times for the Java, ATL, and QVTo transformations.

The transformations have scaled out better for Java and QVT than for ATL. The lowest execution time were for Java in the case of the large-size model, and ATL for the medium-size model. It is worth noting that performance of model transformations can significantly be reduced by applying patterns for a particular language. For example, in the case of ATL for the large-size model, we reduced the running time by half by using the dynamic map pattern previously cited to store the symbol table for variables and exceptions, instead of using `Dict` type attributes in ATL, which are accessed from imperative code sections. ATL supports parallel execution of transformations, and parallel streams of Java 8 can be used to improve the performance of model transformations, but these mechanism have not been explored.

Therefore, the results obtained for development effort and performance would support the use of QVTo against Java or ATL. However, the tools existing for QVTo lack of the maturity offered by commonly used tooling for software development. Software companies might prefer to implement m2m transformation in Java to avoid using immature tools which could be discontinued in the short term. While mature tools are not available for m2m transformation languages, we think that the use of an internal DSL (fluent-API or embedded DSL) could be the more appropriate solution to write model-to-model transformations. In the case of Java, a fluent-API to manage models could be useful. In fact, some tools have been presented with this purpose [39, 40], although these projects were discontinued. A tooling aimed to automate the creation of fluent-API for the APIs that EMF generates could be very valuable to developers writing m2m transformations in Java.

9.3. Developing and testing model transformations

Two essential aspects of the development process here applied are: (i) to write transformations in a incremental way, and (ii) to apply different kinds of tests for validating the transformation chain.

Writing complex transformations for large metamodels requires a systematic approach. The transformations should be created and tested incrementally. For this, we have defined a strategy similar to the test-first programming as explained in Section 5 and illustrated in Figure 8. The transformations are incrementally developed, and a set of input models are created to test each step of its development. That is, the models establish the order in which mappings are implemented and tested. However, tests were not written to validate the transformations. This decision was taken after writing some tests for the *kdm2idioms* transformation. We observe that test code would be very similar to the transformation code to be tested. Below we show the code of the test written to validate if the *kdm2idioms* transformation generates as many "CASE" `SelectionFlows` that have as many `CASE` idioms as `IF ELSIF ELSE` elements there are in the KDM model for the corresponding PL/SQL sentence. JUnit was used to write and run the tests. Although Java 8 facilitates the writing of the transformations and significantly reduces the number of lines of code because the use of streams and lambda expressions, writing tests still requires a considerable effort.

```
for (SelectionFlow selectionFlow :
    selectionFlowList) {
  ActionElement actionElement = selectionFlow.
      getActionElement();
  // Get ELSE
  Optional<ActionElement> posibleElseActionElement
      =
      actionElement.getCodeElement().stream()
          .filter(c -> "ELSE".equals(c.getName()))
          .map(ActionElement.class::cast)
          .findFirst();
  if (posibleElseActionElement.isPresent()) {
    ActionElement elseActionElement =
        posibleElseActionElement.get();
    // Counts ELSIFs
    long elseIfs = elseActionElement.getCodeElement
        ().stream()
          .filter(c -> "IF".equals(c.getName()))
          .filter(ActionElement.class::isInstance)
          .map(ActionElement.class::cast)
          .filter(a -> a.getStereotype()
                          .stream()
                          .filter(e ->"elsif".
                              equals(e.getName().
                              toLowerCase()))
                          .count() > 0)
          .count();
    // Check if ELSE has code
    long elsePresent = elseActionElement.
        getCodeElement().stream()
          .filter(c -> !"IF".equals(c.getName()))
          .count() > 0 ? 1 : 0;
    // IF + ELSE + ELSIFs
    assertEquals(1 + elsePresent + elseIfs,
        selectionFlow.getCase().size());
  } else {
    // IF
    assertEquals(1, selectionFlow.getCase().size())
        ;
  }
}
```

The creation of transformation-specific asserts could facilitate writing tests. However, the implementation of these asserts is a challenging problem since they should be applicable on any metamodel. We consider that the solution proposed in [41] is not practical because XMI format models must be navigated by means of XPath expressions [42]. Therefore, writing unit tests before the transformation code, following the test-first programming of agile methods, seems a practice not suitable in the context of m2m transformations with the existing technology.

With regard to writing unit tests for m2t transformations, we consider interesting the work of Tiso et al. [43]. We could not use their tool, as it is intended to be used on UML with profiles. However, the approach can be adapted writing parameterized regular expressions specific to our models, and the generation of Java code is also suitable for their "sub-transformation" approach following the containment relationships in the target platform model.

We have combined unit tests for a particular model transformation with tests aimed to validate the model transformation chain. As indicated in Section 8, coverage and acceptance tests have been applied along with syntactic and semantic tests on the transformation chain output through out compiler tools.

In model-driven migration scenarios, test models can directly be obtained from source code by using the injector. This is a clear advantage with respect to other model-driven scenarios. Test models can be injected for legacy code or either they can be injected from code specially written for testing. In our case, we have used legacy forms provided by the company to validate the model transformation chain, and have created small code fragments to inject the input models used to validate the first m2m transformation in the chain. The rest of m2m transformations were validated by using the models generated in the precedence transformation according the chain.

9.4. Visualizing output models for testing model transformations

Checking the correctness of the output model of a model transformation is a complicated task due to the complexity of the metamodels involved. Either creating oracles to be used with model comparison tools or performing manual checking are really tedious and time-consuming tasks. Because the difficulty of automatically or manually creating the expected output models, a manual checking is usually performed. Input and output models are examined to verify whether mappings between them are correct. This labor can be facilitated by graphically visualizing output models, but the generation of graphical editors tailored to each metamodel requires a considerable effort. EMF provides a generic tree editor which shows a list of all the model elements and allows navigating over the aggregation hierarchy of each element, but references between objects are not visualized. Examining models with this editor is really difficult.

Because the models are graphs, we have studied how data visualization capabilities offered by some graph databases can be used for visualizing output models. We built a m2t transformation that generates Neo4J insertion script from models representing the execution flow of idioms. These models were finally not used because the execution flow is already represented in the KDM models. The m2t transformation was easy to implement because the execution flow models are graphs whose nodes are statements and the arcs denote all the possible paths that might be traversed during the code execution. Figure 23 shows the execution flow graph for the PL/SQL code below:

```
threshold := 1000;
IF salary > threshold THEN
   bonus := salary - threshold;
ELSE
   bonus := 100 + threshold - salary;
END IF;
salary := salary + bonus;
```

As observed in Figure 23, the node color is used to differentiate the executed trigger (*pink*), the statement fragment (*green*), the initial node (*blue*), and the final node (*purple*). This validation would have been very difficult by using the EMF tree editor. However, it was easily performed through the graph visualization. Moreover, the effort to write the transformation was only of about 4 hours.

This approach of converting models in graph database objects is applicable to many metamodels. It would require to establish a mapping between the model information to be visualized and graph elements. In our case, the only visualization carried out was for the execution flow models, which were used in the final implementation. It might also have been used to show references among the platform model elements. It allows us to check if trigger code has been correctly spread in services (only one method service or more). Another scenario could be to graphically represent the use of variables for each method. In case two or more methods were referring the same variable, then that variable will be tagged as shared. Therefore, it would allow us to visually verify if the algorithm for sharing variables worked. However, the fact of managing many small test models influenced in no implementing more Neo4J-based visualizations.

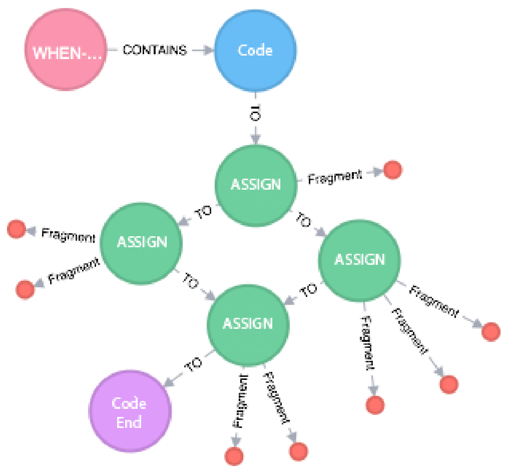


Figure 23: Neo4J graph for a model representing an execution flow.

9.5. Model traceability

Model traceability facilitates the implementation of unit tests in JUnit by taking advantage of the existence of references from Idioms to KDM model elements. It allows us to integrate the developed tool into the one provided by the company (UI2Java). These references enable backward navigation from model elements resulting of transformations involved in the restructuring stage. Therefore, both tools can be connected by means of the generated models.

Model traceability also avoids having to decorate the generated models after each transformation of the tool chain. When the Idioms model was implemented, we decided to keep the back references in order not to add new attributes on the elements for identifying the corresponding element type in KDM.

For instance, in a `SelectionFlow` element on the Idiom model, the reference to the corresponding KDM element provides the needed information about if it was generated by an `If` or an `Switch` KDM element. A specific attribute to identify the corresponding provenance would have been included, but then the Idiom metamodel would had been extremely overloaded until the fact that all the KDM metamodel could be included inside it. It is worth noting that the Idiom metamodel was devised to describe the data at a higher level of abstraction than the KDM metamodel.

10. Related Work

In this work we have presented a practical experience in developing a model-driven reengineering approach for migrating legacy Oracle Forms code to MVC platforms. We have defined a development process that integrated several kind of validation, and present a strategy to implement model transformations in a incremental way. Moreover, an assessment of applying modeling techniques in this scenario is given. In this setting, the works more closely related to our approach would belong to the following categories: (i) model-driven approaches for migrating legacy code, (ii) development processes for MDE solutions, and (iii) model-driven migration experiences. In addition, we also consider studies on the use of KDM, given this OMG metamodel is a key element in our solution.

Before of addressing related works in the above categories, it is worth noting that the state of the art in model-driven reverse engineering has recently been analyzed in [6]. The authors present a systematic literature review in which they describe the more relevant approaches and perform and analysis conducted by some research questions related to the used metamodels and tools, and the level of automation achieved. A total of fifteen proposals are detailed and some hints on choosing a model-driven reverse engineering approach are pointed out.

Model-driven approaches for migrating legacy code.

A model-driven reverse engineering approach to migrate RAD event handlers to modern platforms is proposed in [8]. An AST model is injected from legacy code, and a reverse engineering process is applied to obtain a more abstract representation. Idioms models are firstly extracted, and then an execution flow model is generated. This model represents a graph whose nodes are idioms fragments and the arcs denotes the execution flow. Additionally, each fragment is annotated to indicate which concerns it belongs to: UI, Control or Business Logic. As proof of concept, Ajax code was generated from execution flow graphs. Concretely, PL/SQL event handlers were migrated to a two tier architecture: a HTML/Javascript user interface invokes a REST service that implements business logic fragments. As noted in Section 7, we have used abstracted code into Idiom models, but the reverse engineering here presented differs in two significant aspects from the previous work: (i) KDM is used for representing legacy code, and the execution flow model is not necessary because that information is part of KDM models, (ii) separation of concerns is achieved through a Plaform model, which is used to conduct the object-oriented code generation, and (iii) new idioms have been identified aims to model in KDM all the PL/SQL statements.

In [9] a strategy for reengineering legacy systems to SOA (*Service Oriented Architecture*) is described. The authors consider the decomposition of a monolithic application into several concerns as the main problem to be addressed. They propose manually annotating legacy code to indicate which tier each statement belongs to: logic, data, and user interface. The reengineering process is defined according to the horseshoe model. An AST model is injected from the legacy code, and then a graph-based transformation language is used to redesign the code for the new platform. The authors outline two strategies that they are considering to generate code from the graph model obtained.

Unlike the works of Sanchez et al. [8] and Heckel et al. [9], we have tackled all the stages of a reengineering process. Beyond presenting a reengineering approach, we have defined and applied a development process, which allowed us to explore some key aspects of MDE solutions, such as writing and testing model transformations, and the visualization of models with the purpose of testing. Our reengineering solution has been applied to legacy code from real applications, and have carried out several kinds of testing for validating the result.

Development processes for MDE solutions.

In [11], the authors propose an approach for providing incremental development of model transformation chains based on automated testing. The approach includes four test design techniques along with a framework architecture in order to test transformation chains. The paper also includes a validation of the approach by developing a transformation chain for model version management (for the IBM WebSphere Business Modeler). As it is indicated, doing testing in an isolated way is not sufficient, and the software quality should be assured by means of a development process. They introduce the requirements to ensure the quality of a transformation chain: (i) an iterative and incremental development, (ii) testing processes for the transformation chain, and (iii) a fully automated test environment. The incremental development is based on automated testing, so its approach is supported by a test framework which follows the TDD (test-driven development) principle. The three testing design techniques that the proposal distinguishes are: integrity tests on generated model, comparing the result of a model transformation with inspected reference outcomes, invariant validation, and deviation testing that consists on calculating and storing some data on output models with the purpose of be comparing them with the data obtained when the transformation is again executed.

We have applied an incremental development process similar to that proposed by Küster et al. [11] to develop a reengineering based on the horseshoe model. However, while that process is focused on transformation chains, we have considered how unit tests can facilitate the implementation of individual model transformations. Small input models, which cover one or a few statements of the source language, are used as test models. We have shown how this technique has been applied in our reengineering. Instead, an incremental approach is proposed in [11] but its application is not illustrated with a case study. Conversely, we have performed acceptance tests and unit testing on the generated code. Therefore, we have combined manual and automated testing. An automated test framework for model transformations is desirable but its development is beyond the scope of our work. As far we know, there are no automated testing frameworks for model transformations publicly available today.

A notion similar to the unit tests proposed in our approach was proposed by D. Varró in [12], where the *model transformation by example* technique is presented. This technique aims to semi-automatically generate model-to-model transformation rules from a collection of interrelated input and output models. Each pair of this collection acts as prototypical instance that describes a critical case to be addressed by the transformation. Therefore, our unit test strategy puts into practice a MBTE process. Varró et al. consider that "the majority of model transformations has a very simple structure," and transformations could therefore be partially synthesized. However, we managed a very large metamodel as KDM, and complex transformations, and it is not clear that the statement holds for our case, so and a manual writing and testing has been carried out.

As commented in Section 9, the work by Tiso et al. [43] is applicable to our work as a way of testing model-to-test transformations. In their work, they divide the transformation in sub-transformations, following the containment relationships of the source metamodel. For each transformation, they build parameterized regular expressions that capture the results–in form of text outputs–of this transformation, but ignore the results of the sub-transformations. This allows to effectively associate a regular expression to the production of each of the transformations, allowing a systematic test of the output text. The approach can be used in our development, but their tool could not be used, as it is based on UML and profiles, while we use EMF/Ecore, but a similar approach could be considered in future refinements of the process.

A test-driven development (TDD) approach for model transformation is considered in [26]. Concretely, the JUnit framework is extended in order to facilitate the application of TDD in developing model transformations. Some examples of unit tests are shown for the built prototype. From our experience, writing these unit tests requires a significant effort because they have a large amount of code, and even the correctness of this code should be validated. Therefore, we have not written such unit tests. An interesting future work would be the development of a unit test framework for implementing model transformation in Java. EUnit [27] is an integrated unit test framework for model management tasks that is based on the Epsilon platform. Unit tests are written in the EOL language provided by Epsilon. Writing EOL unit tests was not considered because our transformations were written in Java.

A model-driven software migration methodology is proposed in [44]. This proposal has been validated by means of a case study based on a refactoring of a control software for wind tunnels programmed in C/C++. In the reverse engineering stage a parser is used to generate an annotated parse tree which is translated into XML format. This format is imported into the jABC modeling tool to create customized process graphs to obtain a code model (the control flow graphs of the application). To validate the code models they use a back-to-back test. Firstly, the original source code is restored from the code models by using a code generator, then it is compiled. Thereafter, the restored code undergoes a second reengineering step that includes to restore the source code again from the new code model, and comparing both source codes. In the forward engineering stage, the GUI is analyzed to identify all possible actions of the application. An abstraction model is created with the graphical elements linked to their implementation. The migration and remodeling of the application is a manual task. With the help of the abstraction model, the developer examines the source code to identify and migrate the necessary objects to a new process model. Finally, target code is generated from the process model, but some code is also manually written. The application is validated against the original one by comparing their outputs. The two main differences with our approach are: (i) KDM is not used to represent the legacy code, but control flow graphs obtained by importing the XML code generated by the parser, (ii) the reverse engineering stage is a manual task: The developer examine the code model (control flow graphs) to identify and migrate the necessary objects. At the forward engineering stage some code must be manually written as well, (iii) the process of construction and testing of model transformations is not addressed, and (iv) only a kind of validation is applied: code generated is injected and compared with the refactored source code.

10.1. Model-driven reengineering experiences

In [45], one of the first model-driven reengineering experiences is reported. The work presents the migration of a large scale bank system to the J2EE platform. The three stages of the process are commented and some conclusions are drawn. The model-driven approach is compared to a manual process and some benefits and limitations are provided. However, no details are provided on the development process followed, and the transformation chain is not explained in detail. The models are injected in form of an AST tree.

A model-driven approach for applying a white-box modernization approach has recently been presented in [46]. First, a technology agnostic model is obtained from the sources. Then, this model is edited by the developer to configure the target architecture, and finally a transformation from the model into the new technology is performed. They used an Oracle Forms migration to Java technology case study, and were able to generate the graphical interface (but without layout), the logic related to database operations (e.g. read, update), and the scaffolding code to call the PL/SQL logic that are embedded in the triggers. In contrast to our work, the PL/SQL logic is not transformed and must be manually migrated.

10.2. Use of KDM

As can be noted in [6], a few experiences on the use of KDM have been reported. Below, we comment two of those described in that survey.

Pérez Castillo et al. [47] propose a method for recovering business processes from legacy information systems using MARBLE (Modernization Approach for Recovering Business processes from LEgacy Systems). This method considers (i) a static analysis to extract knowledge from the legacy source code, (ii) a model transformation based on QVT to obtain a KDM model from that knowledge, and (iii) another QVT transformation to create a business process model from the KDM model. MARBLE keeps traceability because it identifies which pieces of legacy code were used to obtain the elements of the business process.

Normantas and Vasilecas [48] present an approach to facilitate business logic extraction from the knowledge about a existing software system. They use various KDM models at different abstraction levels to represent the information extracted from the system in three steps: (i) a preliminary study, to gather information about the system, (ii) knowledge extraction into various KDM representation models, and (iii) separation of the KDM model parts of the business logic from the infrastructure ones. The authors apply source code analysis techniques to identify the business logic and represent it with the KDM Conceptual model.

How ASTM and KDM models can be used to automate a modernization task was addressed in [28]. Some lessons learned on the application of KDM for calculating metrics for PL/SQL code were exposed.

Here, we have detailed discussed complex model transformations involving KDM models, and have discussed some benefits and limitations of KDM from our experience. Moreover, we have experienced the use of stereotypes instead of injecting Micro-KDM compliant models. Both [47] and [48] are not focused on the `Action` and `Code` packages, but business process models are considered. Finally, it is worth noting that we have here addressed a real source-source migration process, while a simpler modernization problem was considered in [28].

11. Conclusions and Future Work

Model-driven techniques emerged early this century as the new software engineering paradigm to achieve levels of productivity and quality similar to other engineering areas. Much attention has been devoted to the application of MDE in engineering scenarios, both in the industry and in the academy. As recently noted in [6], the literature on the application of MDE techniques in model-driven reverse engineering and reengineering is very limited. Practical experiences such as the one described here can be useful to know the benefits and limitations of model-driven techniques in such scenarios. Moreover, they can contribute defining new processes, techniques, and practices, or just experimenting with the existing ones.

In this paper, we have presented a practical experience in designing and implementing a re-engineering approach for a migration of Oracle Forms code to a MVC architecture based on Java frameworks. For this aim, we have defined a systematic process aimed to the development of the model transformation chain implementing the reengineering. This process involved several kinds of tests for validating the model transformations and the generated code. The definition of this process and the description of its applications is a contribution with respect to previously published model-driven reengineering approaches. Specially, we would remark the strategy applied to incrementally develop model transformations.

Regarding the benefits of applying MDE in a software modernization scenario, it is well known that (i) metamodels provide a formalism more appropriate than other metadata formats (e.g. XML, JSON) to represent information harvested in applying reverse engineering, and (ii) transformations allow migration tasks to be automated [4, 6]. In this work, we also contribute with knowledge about some specific concerns in a software migration. The main contributions would be the following.

Use of KDM. Experimenting with KDM in a real project, and showing that L1 compliant KDM models for data and Micro-KDM are very convenient to represent code from a structural and behavioral viewpoint. Thanks to the usage of KDM, we did not need to define a PL/SQL, Data or execution flow metamodels. In addition, KDM provides a code and data representation devised by experts in migration. This a clear example on the advantages of asset reusing.

Writing model transformations. We have compared Java with two of the most widely used languages for writing model-to-model transformations. This comparison was performed for one of the transformations of our work. We have concluded that a fluent-API for Java (backed by EMF/Eclipse as the more widespread modeling framework) is the more appropriate choice until mature and robust environments for model-to-model transformation languages are available.

Models for testing. Creating input and expected output models for validating model transformations is very difficult as noted in [23]. We have shown that software migration makes it easier to have input models as they can be achieved from source code by using the model injector. On the other hand, the model transformation chain generates code. Checking the correctness of this code, and its correspondence with the original code to be modernized, is again a very difficult task. The usage of a test-first, incremental transformation development helped in checking each of the mappings from models to text. Other approaches were also considered and left as future work [43]. It is worth noting that we have investigated the transformation of models into instances of graph databases in order to take advantage of viewers and query languages offered by these database systems. This would provide a simple strategy to visualize output models of a transformation, which would facilitate the manual validation.

The practical experience here presented has served to define some research works for some issues here raised, such as: (i) extend JUnit to develop a automated unit test framework for model transformations, (ii) explore test-driven development for model transformations, (iii) complete the comparison among transformation languages, (iv) integrate *Models4Migration* [49] with the model management platform of Open Canarias, and (v) to define a systematic approach to visualize models as graph databases and explore the benefits of this representation for automating different kinds of testing (e.g. coverage testing).

12. References

[1] S. R. Tilley, D. B. Smith, Perspectives on legacy system reengineering, Tech. rep., Software Engineering Institute, Carnegie Mellon University (1995).

[2] R. Kazman, S. S. Woods, S. J. Carrière, Requirements for integrating software architecture and reengineering models: CORUM II, in: 5th Working Conference on Reverse Engineering, WCRE '98, Honolulu, Hawai, USA, October 12-14, 1998, 1998, pp. 154–163. doi:10.1109/WCRE.1998.723185.
URL https://doi.org/10.1109/WCRE.1998.723185

[3] F. J. Bermudez, J. G. Molina, O. Díaz, On the application of model-driven engineering in data reengineering, Inf. Syst. 72 (2017) 136–160.

[4] O. Sánchez Ramón, J. Sánchez Cuadrado, J. García Molina, Model-driven reverse engineering of legacy graphical user interfaces, in: Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10, ACM, New York, NY, USA, 2010, pp. 147–150. doi:http://doi.acm.org/10.1145/1858996.1859023.
URL http://doi.acm.org/10.1145/1858996.1859023

[5] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Synthesis Lectures on Soft. Eng., Morgan & Claypool Publishers, 2012.

[6] C. Raibulet, F. A. Fontana, M. Zanoni, Model-driven reverse engineering approaches: A systematic literature review, IEEE Access 5 (2017) 14516–14542. doi:10.1109/ACCESS.2017.2733518.
URL https://doi.org/10.1109/ACCESS.2017.2733518

[7] O. M. Group, Knowledge Discovery Meta-Model (KDM), document formal/2011-08-04. (2011).
URL http://www.omg.org/spec/KDM/1.3

[8] O. Sánchez Ramón, et al., Reverse engineering of event handlers of rad-based applications., in: M. Pinzger, D. Poshyvanyk, J. Buckley (Eds.), WCRE, IEEE Computer Society, 2011, pp. 293–302.
URL http://dblp.uni-trier.de/db/conf/wcre/wcre2011.html

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

[10] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, Y. Le Traon, Model transformation testing challenges, in: Proceedings of the IMDDMDT workshop at ECMDA'06, Bilbao, Spain, 2006.
URL http://www.irisa.fr/triskell/publis/2006/baudry06b.pdf

[11] J. M. Küster, T. Gschwind, O. Zimmermann, Incremental development of model transformation chains using automated testing, in: Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings, 2009, pp. 733–747. doi:10.1007/978-3-642-04425-0_60.
URL https://doi.org/10.1007/978-3-642-04425-0_60

[12] D. Varró, Model transformation by example, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2006, pp. 410–424.

[13] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Sci. Comput. Program. 72 (1-2) (2008) 31–39. doi:10.1016/j.scico.2007.08.002.

[14] OMG, Query/View/Transformation. Object Management Group (OMG) (2011).

[15] O. M. Group, Abstract Syntax Tree Meta-Model (ASTM) (2011).
URL https://www.omg.org/spec/ASTM/

[16] O. M. Group, Structured Metric Meta-Model (SMM) (2016).
URL https://www.omg.org/spec/SMM/

[17] OMG, Automated function points specification (afp) (2014).

[18] ATo ms, Página de la especificación de User Interface Extended Markup Language, http://www.usixml.org/en/what-is-usixml.html?IDC=236, Último acceso: 23-06-2017.

[19] O. M. Group, Página de la especificación de Interaction Flow Modeling Language, http://www.omg.org/spec/IFML/1.0/, Último acceso: 23-06-2017.

[20] JCP, JSR-000344 JavaServer Faces 2.2 Final Release, https://jcp.org/aboutJava/communityprocess/final/jsr344 (2013).

[21] JCP, JSR-000338 Java Persistence 2.1 Final Release, https://jcp.org/aboutJava/communityprocess/final/jsr338 (2013).

[22] E. J. Chikofsky, J. H. C. II, Reverse engineering and design recovery: A taxonomy, IEEE Software 7 (1990) 13–17.

[23] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, J. Mottu, Barriers to systematic model transformation testing, Commun. ACM 53 (6) (2010) 139–143. doi:10.1145/1743546.1743583.
URL http://doi.acm.org/10.1145/1743546.1743583

[24] F. Fleurey, J. Steel, B. Baudry, Validation in model-driven engineering: testing model transformations, in: Model, Design and Validation, 2004. Proceedings. 2004 first international workshop on, IEEE, 2004, pp. 29–40.

[25] A. Ciancone, A. Filieri, R. Mirandola, Mantra: Towards model transformation testing, in: Quality of Information and Communications Technology, 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010, Porto, Portugal, 29 September - 2 October, 2010, Proceedings, 2010, pp. 97–105. doi:10.1109/QUATIC.2010.15.
URL https://doi.org/10.1109/QUATIC.2010.15

[26] M. J. McGill, B. H. Cheng, Test-driven development of a model transformation with jemtte.

[27] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, I. Medina-Bulo, Eunit: A unit testing framework for model management tasks, in: Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings, 2011, pp. 395–409. doi:10.1007/978-3-642-24485-8_29.
URL https://doi.org/10.1007/978-3-642-24485-8_29

[28] J. L. C. Izquierdo, J. G. Molina, An architecture-driven modernization tool for calculating metrics, IEEE Software 27 (4) (2010) 37–43. doi:10.1109/MS.2010.61.

[29] H. Bruneliere, J. Cabot, G. Dupé, F. Madiot, Modisco: a model driven reverse engineering framework 56 (2014) 1012–1032.

[30] Eclipse, Acceleo (2012).
URL http://www.eclipse.org/acceleo/

[31] OMG, MOF Model to Text Transformation Language (MOFM2T), 1.0, http://www.omg.org/spec/MOFM2T/1.0/ (2008).

[32] M. U. Guide, Gastm overview, https://help.eclipse.org/neon/index.jsp?topic= accessed: April 2018.

[33] C. J. Fernández, J. R. Hoyos, J. García-Molina, F. J. Bermúdez, B. J. Cuesta, Una experiencia en la implementación del método afp, in: Proceedings of the XVII JISBD, 2017.

[34] Ó. Sánchez, J. S. Cuadrado, J. G. Molina, J. Vanderdonckt, A layout inference algorithm for graphical user interfaces, Information & Software Technology 70 (2016) 155–175.

[35] D. Kolovos, R. Paige, F. Polack, The epsilon transformation language, in: A. Vallecillo, J. Gray, A. Pierantonio (Eds.), Theory and Practice of Model Transformations, Vol. 5063 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 46–60. doi:10.1007/978-3-540-69927-9_4.

[36] J. Sánchez Cuadrado, J. García Molina, M. Menárguez Tortosa, RubyTL: A Practical, Extensible Transformation Language, 2006.

[37] E. Jakumeit, et al., A survey and comparison of transformation tools based on the transformation tool contest, Sci. Comput. Program. 85 (2014) 41–99.

[38] L. M. Rose, et al., Graph and model transformation tools for model migration - empirical results from the transformation tool contest, Software and System Modeling 13 (1) (2014) 323–359.

[39] M. Garcia, dsl2jdt, http://www.eclipse.org/articles/Article-AutomatingDSLEmbeddings/.

[40] B. Fagin, Flapi, https://github.com/UnquietCode/Flapi, visited 2018, April.

[41] M. J. McGill, B. H. Cheng, Test-driven development of a model transformation with jemtte, Tech. rep., Michigan State University (2007).

[42] W3C, XPath XML Path Language 3.1 (W3C) (2017).

[43] A. Tiso, G. Reggio, M. Leotta, Unit testing of model to text transformations, in: Proceedings of 3rd Workshop on the Analysis of Model Transformations (AMT 2014 co-located with MoDELS 2014), Vol. 1277, 2014.

[44] C. Wagner, Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems, Springer Science & Business Media, 2014.

[45] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, J. Jézéquel, Model-driven engineering for software migration in a large industrial context, in: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings, 2007, pp. 482–497. doi:10.1007/978-3-540-75209-7_33.
URL https://doi.org/10.1007/978-3-540-75209-7_33

[46] 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.

[47] R. Pérez-Castillo, I. G. R. de Guzmán, M. Piattini, Business process archeology using MARBLE, Information & Software Technology 53 (10) (2011) 1023–1044. doi:10.1016/j.infsof.2011.05.006.
URL https://doi.org/10.1016/j.infsof.2011.05.006

[48] K. Normantas, O. Vasilecas, Extracting business rules from existing enterprise software system, in: Information and Software Technologies - 18th International Conference, ICIST 2012, Kaunas, Lithuania, September 13-14, 2012. Proceedings, 2012, pp. 482–496. doi:10.1007/978-3-642-33308-8_40.
URL https://doi.org/10.1007/978-3-642-33308-8_40

[49] F. J. B. Ruiz, Ó. S. Ramón, J. G. Molina, A tool to support the definition and enactment of model-driven migration processes, Journal of Systems and Software 128 (2017) 106–129. doi:10.1016/j.jss.2017.03.009.
URL https://doi.org/10.1016/j.jss.2017.03.009