Applying Model-Driven Engineering in Small Software Enterprises

Jesús Sánchez Cuadrado[a,*], Javier Luis Cánovas Izquierdo[b], Jesús García Molina[c]

[a] *Universidad Autónoma de Madrid (Spain)*
[b] *AtlanMod, Ecole des Mines de Nantes INRIA, Nantes, France*
[c] *Universidad de Murcia (Spain)*

Abstract

Model-Driven Engineering (MDE) is increasingly gaining acceptance in the software engineering community, however its adoption by the industry is far from successful. The number of companies applying MDE is still very limited. Although several case studies and reports have been published on MDE adoption in large companies, experience reports on small enterprises are still rare, despite the fact that they represent a large part of the software companies ecosystem.

In this paper we report on our practical experience in two transfer of technology projects on two small companies. In order to determine the degree of success of these projects we present some factors that have to be taken into account in transfer of technology projects. Then, we assess both projects analysing these factors and applying some metrics to give hints about the potential productivity gains that MDE could bring. We also comment on some lessons learned. These experiences suggest that MDE has the potential to make small companies more competitive, because it enables them to build powerful automation tools at modest cost. We will also present the approach followed to train these companies in MDE, and we contribute the teaching material so that it can be used or adapted by others projects of this nature.

Keywords: Model Driven Engineering, experience report, small companies, incremental consistency

1. Introduction

Model Driven Engineering (MDE) has emerged as a new software engineering discipline which emphasizes the use of models to improve the software productivity and some aspects of the software quality such as maintainability or interoperability. MDE techniques have proven useful not only for developing new software applications but for reengineering legacy systems and dynamically configuring running systems. Several software development paradigms are included in MDE, among which Model-Driven Architecture MDA [1], domain-specific development [2], language-oriented programming [3] and model-driven modernization [4] have received more attention. As noted by [5], the different MDE paradigms can be reduced to two main ideas: raising the level of abstraction and raising the degree of computer automation.

Although MDE is increasingly gaining acceptance in the software engineering community, its adoption by the industry is very limited. There are a growing number of companies which have successfully applied MDE [6], but "its use by the industry is still an exception rather the norm" [5]. Indeed, the industrial adoption of a software innovation is not immediate, but a period of time (more than two decades in the case of object technology) is required to reach the needed conditions: a stable and mature foundation, usable and robust tools, skilled professionals and companies aware of the benefits of the new technology. Therefore, three main actions must be carried out for MDE to be successful: more research and development effort to overcome the technical challenges; including MDE in the

*Corresponding author
Email addresses: jesus.sanchez.cuadrado@uam.es (Jesús Sánchez Cuadrado), javier.canovas@inria.fr (Javier Luis Cánovas Izquierdo), jmolina@um.es (Jesús García Molina)
URL: http://sanchezcuadrado.es (Jesús Sánchez Cuadrado), http://jlcanovas.es (Javier Luis Cánovas Izquierdo)

University curricula; and projects of Transfer of Technology (ToT) intended to increase the awareness of industry, as well as training software developers in the model-based style of thinking [7].

Most experience reports about MDE adoption are focused on large companies, however as noted in [8], "small companies are different" as they have some particularities that cannot be overlooked. Both large and small companies face similar software engineering challenges, but the solutions have to be adapted to the size and nature of the company. For instance, small companies are more responsive and flexible, but they do not typically have enough resources to build custom, in-house solutions. In this way, the adoption of MDE in a small company has to be different from a large one. It is particularly important to scale well the initial MDE projects. For instance, it is not reasonable to convert the code-centric development style of a small company into a model-centric style, since the cost is high, and the benefits have not been clearly assessed yet [9]. Instead, a better alternative is to use MDE techniques to automate certain development problems as a means to enhance the company productivity.

In this work we report on our practical experience in two ToT projects. Unlike most experience reports, we have dealt with two small software companies: Sinergia with around 100 employees, with which we carried out a modernization project, and Visualtis with 12 employees (at the time of the project) which built a generative architecture. The analysis of these experiences suggest that MDE has the potential to make small companies more competitive, because it enables them to build powerful automation tools at modest cost. By reporting on these projects, our hope is to give insights that may help other ToT projects aiming at introducing MDE to small software companies. To this end, we introduce some factors that affect the success of this kind of projects, and provide an assessment of both projects on the light of these factors. The assessment is based on qualitative data about the projects, and quantitative data in the form of metrics. We also present some lessons learned. In addition, the description of each pilot project may serve as an inspiration for small companies that want to begin with MDE on their own, since they are prototypical small-medium projects that can be addressed without spending much resources and pay off rapidly. We will also present our approach for training companies in MDE, and we contribute the teaching material so that it can be used or adapted by other ToT projects.

Paper organization. Section 2 introduces the factors considered in assessing the success of our ToT projects. Sections 3 and 4 describe in detail the two projects: generating Java wrappers and building of a generative architecture. Section 5 gives an assessment of the projects by analyzing the factors previously introduced, while Section 6 discusses some lessons learned from these experiences.

2. Success factors in ToT projects

Our research team has always been conscious of the need of collaborating with software companies in order to apply the research results obtained. Since we started to work in MDE in 2005, we have participated in several ToT projects which have been funded by the national or regional government with the aim of promoting the innovation and research in regional industries. Based on this experience we have identified several factors that affect the development of ToT projects, particularly in small companies. We introduce them in the rest of this section, but first we discuss our view about success in this kind of projects.

2.1. Success in ToT projects

A ToT project is aimed at transfering research results (normally knowledge, technologies, skills or methods) from research institutions (e.g., universities) to companies. Determining the degree of success in this kind of projects must be done according to the objective of the transfer. In our case, both projects shared a common objective: to show the benefits of MDE technology by building a tool able to automate some software development tasks in the companies. Therefore, the experience would serve to the company's staff to realize the potential of MDE to improve their productivity by automating certain tasks. From this perspective, the sucess of our projects should be assessed from three main dimensions: 1) at the knowledge dimension (i.e., to what extent the company has acquired an accurate judgement on the potential and current state of the technology), 2) at the product dimension (i.e., the application of MDE techniques for building the desired tool has proved useful) and 3) at the adoption dimension (i.e., is the company committed, somehow, to adopt MDE in some form in the future?). It is important to note that the fulfilment degree of each dimension may vary according to the goals or expectations of the project. In this way, some projects may be mainly focused on transferring a cutting edge technology, and evaluate a possible adoption in a future, as is the case of our projects.

Knowledge dimension. This dimension is intended to make a company aware of the features and current status of a given technology, and ideally with the knowledge needed to apply the technology without further help on the behalf of the university.

Providing training to company developers, and building a tool for the company in order to the advantages of the new technology are common activities to accomplish a transfer of knowledge. Both approaches can be combined as we have done in one of our projects.

On the other hand, a ToT project can also serve as an opportunity for the university to realise the real needs of industry [10]. Which is more, synergies could arise that enable the application of established industrial techniques to solve research problems.

Product dimension. A ToT project can involve building a product by applying the new technology. In the case of MDE, building a tool aimed at solving a certain problem of the company is a good option to show how MDE techniques can improve the productivity of the software company. This is particularly important for small companies, since powerful automation tools can be built at modest cost.

To this end, scoping the tool being built it is quite important: the tool has to be finished and tested within the project life span, and the company should have the impression the devoted resources has been adequate.

Adoption dimension. In an ideal case, the acquired knowledge helps the company adopt the new technology in the medium-term if it has been found suitable. Unlike the previous dimensions, the adoption of a new technology may also be influenced by non-technical aspects such as ethics and moral of the company [11, 12], and social and organizational aspects [6], which may hamper its degree of achievement.

In the case of small companies, it is not realistic to demand a conversion from the code-centric development style into a model-centric style. Instead, it would be more reasonable to expect that the company, as a result of the ToT project, considers MDE as a useful technology that can be used ocassionally to address tasks that can be automated, as a way to boost productivity.

As stated before, determining whether these dimensions have been successful depends on the initial goal of the project and the final degree of achievement.

2.2. Success factors

In the following we comment on several factors which have to be considered to assess the degree of achievement of transference and adoption in a ToT project. These factors will be used in Section 5 to assess the two projects presented in Sections 3 and 4.

2.2.1. Industrial context and innovation

This factor refers to the industrial development degree of the geographical region in which the project is carried out. It is also important to consider the innovation culture: the more companies invest on innovation the easier is to develop a ToT project since they are open to try new technologies. However, a low level of R&D+i (Research, Development and Innovation) activities strongly hinders the adoption of new technologies, MDE in this case.

In our experience, a context of low innovation culture requires an additional effort on the behalf of the university to search for software companies interested in applying MDE, rather than receiving proposals from software companies betting on innovation. In our case we typically organized a presentation at the company in order to briefly introduce MDE technology and show, through some examples, the potential benefits that MDE may bring to them. Normally both project managers and developers were initially attracted by the possibilities of MDE technology for automating software development, although they were skeptical about its applicability in real projects (in [13] similar problems are reported). In these meetings, we have always been realistic, recognizing the limitations and immaturity of MDE but we have highlighted the opportunity in gaining a competitive advantage at low cost and risk. After this presentation, we met with project or company managers to know both the development processes applied and the kind of applications built at the company. Our aim was to harvest some candidate tasks to be automated using MDE techniques.

2.2.2. Size and dynamics of the company

As we have commented in the introductory section, the size of the company is an important aspect to take into account. However, the notion of large, medium or small company varies from country to country. For instance, in the European Union it is considered that a small company has fewer than 50 employees while a medium-sized has fewer than 250 employees [14]. In legal terminology, all of them are named Small and Medium-sized Enterprises (SME). Nevertheless, in practice there are differences between a company with, e.g., 250 employees, and a company with 100 employees. In this paper we will consider that a company with less than 100 employees is small, since in our experience the resources that they can devote to learning and applying new technologies are also quite limited. As we will see, the flexibility and adoption culture is lower than in an actual small company (e.g., 12 employees in the case of Visualtis).

2.2.3. Style of ToT project

There are two main ways of developing a ToT project: university-centric or company-centric. This has a certain impact on the expected outcome of the project.

In a *university-centric* project, the research team leads the development of the project, which includes hiring engineers to build the product. Communication between university and the research team is needed to ensure that the product being built is what it is expected. The project can be considered successful if the product proofs to be useful and it is integrated in the development process (product dimension), and if from this experience the company realizes the benefits of the technology being introduced (knowledge dimension). Regarding the adoption dimension, it could be considered sucessful if the company decides to carry out further projects, this time in a company-centric setting.

In a *company-centric* setting the product is built by developers of the company. In some cases, the project funds are invested in hiring one or more new developers for the project, but this is far from ideal, since they are not involved yet in the company dynamics. This kind of project typically requires some training. Thus, the research team would be in charge of the training, then the company will start the project on the basis of the acquired knowledge, but it will be the responsibility of the research team to keep track of the progress of the project and to give advice regarding the usage of the technology under consideration. The project will be successful from a if the company team is able to develop the desired product, it is useful to automate the planned tasks, and it is integrated in the development process (product dimension). If this is the case, they have probably acquired the capabilities to tackle other projects on their own (knowledge dimension). Regarding the adoption dimension, it will be successful if the company decides to tackle new, probably small, projects using the technology, either on its own or asking for some advice to the university.

2.2.4. Nature of the product

The nature and scoping of the tool built to demonstrate the potential of MDE must be carefully selected, as stated above. In our case we normally try to identify these prototypical situations: manual tasks which were performed following a recipe (as the two projects described in this paper); a framework whose application could be automated by defining a DSL; and a product family for which a software generative architecture could be created.

2.2.5. Qualifications and training of the developers

The underlying nature of MDE requires certain skills for the developers to be able to learn its foundations and to apply it. Notably, some knowledge about domain modeling (e.g., using UML) and expertise in object-oriented programming is very desirable.

Another important factor is the kind of training given to the developers. This has an influence on the time required for developing the MDE product. A good training is likely to lead to a shorter development time, a better product and also increasing confidence in MDE.

2.2.6. Productivity gains

Productivity gains are normally given as one of main factors for success. Even though the application of a MDE tool in practice gives some idea about the productivity improvement (e.g., generating code is easier than writing it), it is important to have a more precise intuition. Thus, in order to assess the productivity gains, we propose a threefold analysis:

1. Measure the implementation effort of the tool developed in the project. We will present the time involved in developing each artefact (meta-model, grammar, transformations) and the lines of code (LOCs) for grammars and transformations. We accompany LOCs with grammar metrics that give a hint about the complexity of the languages used to develop such artifacts. In some way we are using these metrics as a way to normalize LOCs.
2. Compare the effort of the traditional approach (i.e., manual approach) against the effort of using the created MDE tool. To this end we have also applied grammar metrics to determine the complexity of the languages involved in the manual approach and the automatic approach (e.g., using a DSL). This will also give us some hints about facility to comprehend a system created using the model-based vs. the manual approach.
3. Calculate the return of investment (ROI). Building a MDE tool to automate a repetitive task is necessarily more costly than writing this repetitive code manually once or twice. This means that companies interested in MDE have to aware that the solution they build has to be used in a certain number of projects in order to have a ROI.

These measurements will be presented in Section 5, but the metrics are presented here. For metamodels, we will measure the effort counting the number of metaclasses, attributes and references. When a textual language is involved we will use LOCs, and a set of grammar metrics [15]:

- *HAL*, which calculates the Halstead effort and evaluates grammar designer's efforts to understand the grammar. High values of HAL mean a high effort to understand the grammar.
- *LRS*, which represents the number of states in the LR automaton derived from the language grammar. This metrics captures the complexity of the grammar. DSL grammars have smaller values than GPL grammars (less than 1000 for DSLs).
- *LAT/LRS*, which is a normalized metrics computing the complexity of the relations between the terminals and the states in LR tables. A low value in this metrics means that each terminal only appears in few of the grammar's states and thus the language is very controlled and probably easy to learn.
- *SS*, which builds, for each production, the shortest sample that uses it and stores the average size of these samples. This metrics provides hints of the verbosity of the language produced by the grammar. The higher the value the more verbose the language.

These metrics will be used in the following way. To measure the implementation effort of a grammar (e.g., the textual concrete syntax of a DSL) we will use the HAL metric (the LRS metric could be used as well). We will also use the LRS, LAT/LRS and SS metrics applied to the grammar of the language used to develop such a grammar (e.g., Xtext to develop a textual concrete syntax). This will give us a hint about the complexity of using such a language. In a similar way, we will apply the LRS, LAT/LRS and SS metrics to the languages used for model transformation (e.g., MOFScript) to allow us to understand the complexity involved in the LOCs developed using such a language.

In order to compare the manual and automated approaches, the strategy will be as follows. We will measure LRS, LAT/LRS and SS for the language(s) used in the manual approach (e.g., Java). The same metrics will be measured for the DSL used as input of the generator (if applicable). This will provide an intuition about how difficult is to develop this manual code, against just using the DSL and let the generator create the code automatically. For instance, differences in LRS and LRS/LAT could indicate which approach is more comprehensible (i.e., if one language is easier to understand and learn than another).

3. Modernization project

3.1. Company details

The first project was carried out along with Sinergia Tecnológica, a company with around 100 employees, but nevertheless one of the biggest enterprises in our region.

Sinergia is a company specialized in consulting and IT services, founded in Murcia in 1995. The company has expertise in several areas of software and systems, but notably in the development and maintenance of customized software and IT outsourcing (integrated service or process management in information technology). Around 65% of its business is devoted to the public sector and 35% to the private sector. In the case of the public sector, much of its business is related to the regional government, either in the form of IT outsourcing or by developing new projects.

3.2. Project setup

This project aimed at automating a manual programming task which was identified by our team in one meeting with a project leader of Sinergia. He explained us that the company was involved in the modernization of Forms applications to the Java platform, which required writing a large amount of Java code to wrap business logic code written in PL-SQL, by following a recipe they had elaborated.

We proposed to the company the creation of a tool intended to totally automate the creation of these wrappers. This task was considered appropriate for the company to acquire knowledge on the capabilities of MDE for automating software development, and they contracted us as part of a funded ToT project.

In this way, the development of the tool was performed by two engineers, hired to work at the university facilities, under the supervision of the members of the research team. They were postgraduate students with some previous, informal knowledge in model-driven development, although they were not experts at that time. By the company side, a chief developer was in charge of validating the tool, running tests and assisting us with their knowledge about the problem. The collaboration and project tracking was carried out by setting up a web-based project management system. In particular, Trac [16] was chosen, which features a ticket system for tracking tasks and bug reports, a wiki and integration with Subversion.

In order to develop the tool an iterative strategy was followed. Each released version was tested and validated by the chief developer of the company, which in turn shared the tool with the rest of the company developers to detect bugs and problems. Such issues were reported mainly by the Trac system by means of tickets, which helped us to control the evolution and improvement of the tool. The Trac system also allowed us to provide a website where some documentation (i.e., manual and tutorials) was published.

3.3. Problem description

Sinergia had a large trajectory building and maintaining systems based on Oracle Forms, a fourth generation environment very popular some years ago. Given this experience, Sinergia followed the good practice of developing the business logic of the application as stored procedures that were kept in the database server and invoked by the client (instead of polluting the GUI code with business logic, which is unfortunately common practice in Oracle Forms). These coding rules and guidelines allowed therefore developers to create Oracle Forms applications faster as well as promoting maintainability and legibility.

However, as web applications gained popularity, the Oracle Forms architecture became obsolete. In 2007, Sinergia was involved in a project to create a new version, based on a Java web platform, of an existing Oracle Forms application. The project had the requirement that the business logic already developed as PL/SQL stored procedures should not be migrated but integrated with the new Java codebase, so that the Oracle Forms version and Java web version of the application could co-exist. This involved, for each use of a stored PL/SQL procedure, developing Java code that should be in charge of performing three actions: i) using JDBC to connect to the database server, ii) invoking the stored procedure, and iii) performing datatype conversions from Java to PL/SQL and vice versa. In this way, the company decided to build Java wrappers to hide this complexity.

In this context, for each PL/SQL package a Java class which encapsulated the access to PL/SQL functions and procedures was created (for the sake of clarity we will only use the procedure term except if a distinction is needed). This was done by creating a Java method for each procedure of the package. However, the correspondences between PL/SQL procedures and Java methods differed according to a set of non-trivial rules developed by the company, based on names, types or even the location of the element in the source code. These rules are summarized next:

1. *Methods*. A Java method must be created for each PL/SQL procedure. These methods must be created in the same order as the corresponding procedure appears in the PL/SQL code, but considering grouping as described by rules 9 and 10.
2. *Calling constants*. A Java constant must be created for each PL/SQL procedure, which will be used by the corresponding Java method to invoke the PL/SQL procedure. This constant is a string which specifies the PL/SQL native code needed to call the PL/SQL procedure by using the JDBC API.
3. *Output parameter constants*. A Java constant must be created for each parameter of type `OUT` in the PL/SQL procedure. The name of this constant will be `PAR_<VariableName>` and its value will be the name of the variable. Moreover, if a function is being dealt with, a constant named `PAR_RETURN_OUT` will also be created.

4. *Filtered procedures*. Only PL/SQL procedures which use `VARCHAR2`, `NUMBER`, `BOOLEAN`, `CLOB` and `XML` parameters are mapped, if they use another type, they are discarded. Those procedures which have `INOUT` (input/output) parameters will not be mapped either.
5. *Name-based type inference*. In PL/SQL, the `%TYPE` operator is used to let the type of the variable unspecified until runtime (i.e., to enable dynamic typing). However, the company coding rules allow inferring the type statically by analyzing the variable name, in particular, the first letter of the variable name indicates the data type. Thereby, when the letter is `v`, the type will be `VARCHAR2`, `n` for `NUMBER`, `b` for `BOOLEAN`, `c` for `CLOB` and `x` for `XML`. For instance, the variable `bHaveBalance%Type` is a boolean variable.
6. *Special function: exists*. Functions named `exists` are mapped into two Java methods: the one corresponding to rule 1 and another one which returns a connection.
7. *Special function: haveChildren*. Functions named `haveChildren` must return a `AnswerHaveChildren` object.
8. *Standard functions*. Any function without parameters and located between an `exists` function and a function whose name starts with `have` are called *standard functions*. The corresponding Java method must have an additional parameter to receive the JDBC connection already opened.
9. *Procedure to method mapping*. The mappings between procedures and methods depend on the relative position of the procedure within the PL/SQL file to which the procedure belongs to. In this way, the PL/SQL code packages are implicitly divided into four consecutive zones, which are the following:
   (a) a set of procedures whose name does not begin with `exists` or `have`, until the first one is encountered (called *basic zone*),
   (b) a set of procedures whose name begins with `exists` (called *exists zone*),
   (c) a set of standard functions (called *standard zone*), as described in rule 8,
   (d) a set of procedures whose name begins with `have` followed by the rest of procedures (called *basic zone* as well).
10. *Generated code organization*. The generated Java class must be divided into five zones, which corresponds to the zones of the previous rules plus an initial zone with constants (call and out parameter constants). In addition, the generated code must deal with overloading.

The task of creating these wrappers was performed manually and its automation was considered appropriate to introduce MDE technology in the company. It was an interesting ToT project because the resulting tool could be integrated in the project the company was carrying out at that time, which would help us show them that MDE could improve their productivity.

3.4. Example

To illustrate the problem of creating wrappers for PL/SQL code, Figure 1 shows an example of a Java method to wrap the call to a PL/SQL function following the rules presented previously. For the sake of clarity, the identifiers and method names have been translated into English. According to rule 1, a Java method is created for the PL/SQL function. In the Java code, note the generation of the constant `GIVEMELASTVERSION` to invoke the PL/SQL function (rule 2). The constants `PAR_VDESCPRO` and `PAR_RETURN_OUT` are also generated for the `OUT` parameter and return value (rule 3). Note that the type of the variable `vDescPro` can be inferred as a `Varchar` according to rule 5.

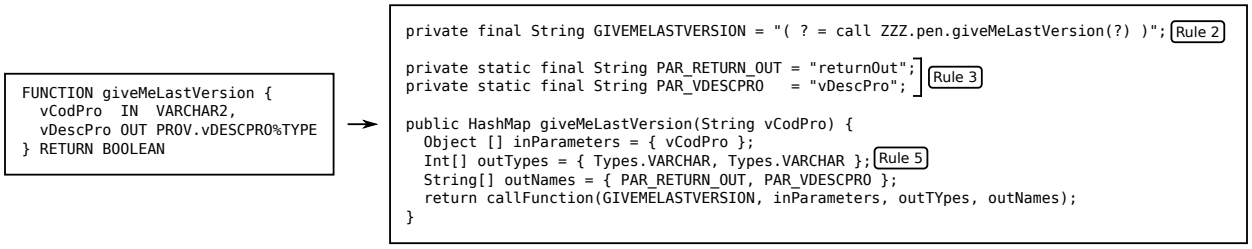


Figure 1: Example of wrapper method generated according to the company rules

3.5. Solution: Automating wrapper generation

Wrapping is a well-know modernization technique, which is often applied as a form of black-box modernization so that developers only have to deal with system interfaces but not with system internals [17]. However, Sinergia's wrappers are an example of a white-box modernization case due to the rules explained before.

In this context, software reengineering is a form of modernization that improves capabilities and/or maintainability of a legacy system by introducing modern technologies, and it provides a systematic approach to migrate a legacy system. A reengineering process involves three phases: i) reverse engineering of the legacy system, ii) redesign of the new architecture, and iii) generation of the target system [18]. As explained in [19], these tasks can be tackled by using MDE techniques such as meta-modeling and model transformation, once models are injected from the legacy code as first step. Automating the generation of wrappers from legacy code is good example of a software reengineering process than can be automated by applying MDE techniques. In this particular case, MDE promotes interoperability among tools (e.g., model transformation languages) and provides a more systematic solution, since it is centered around a language metamodel.

In this project, the reverse engineering phase consisted of obtaining models from existing PL/SQL code, which involved the definition of a PL/SQL injector (i.e., an artefact to parse PL/SQL files and generate models) and the corresponding metamodel. The redesign phase had already been carried out by the company, by devising the set of rules to build the Java wrappers. The creation of the target artefacts consisted of the generation of the Java wrappers from the injected PL/SQL models. To this end, we devised two solutions. At the very beginning, we decided to apply a single model-to-text transformation to generate the wrappers with the aim of avoiding overcomplicating the generation process because, at that time, we were not fully aware the rules explained above. Therefore, it was thought that it would be relatively easy to perform the generation phase in only one step. This is a recommended practice when the semantic gap is small [2, 20]. As the project evolved, the rules and constraints to generate the Java code turned very specific from the company, which caused a great number of modifications in the model-to-text transformation as we explain below, thus hindering its maintenance, readability and evolution. Then, we decided to assess the combination of using model-to-model and model-to-text transformations.

Both cases shared a relatively simple architecture, which just required developing three main artefacts, a PL/SQL injector, an abstract syntax metamodel for PL/SQL and transformations to generate wrappers (using a direct model-to-text approach or going throgh an intermediate model-to-model step). They are explained next.

Creating a PL/SQL injector. In the problem at hand, it was not necessary to deal with the complete PL/SQL language to inject models from the code. Instead, it was enough to process package declarations, which just declare the signature of the stored procedures which had to be wrapped. This fact simplified the injection step, and allowed us to use Xtext, one of the most mature tools which existed at that moment for defining textual DSLs.

As described in [21], although Xtext is mainly aimed at defining textual DSLs, it can also be used for injecting models from legacy code conforming to a General Programming Language (GPL) grammar when such a grammar (or a subset) is not too complex. To do so, the GPL grammar must be expressed in the notation provided by Xtext to define DSLs. From this grammar specification, Xtext is able to generate both the abstract syntax metamodel and a parser to obtain models conforming to such metamodel (i.e., the injector).

Abstract syntax meta-model. Once the Xtext grammar was defined, we automatically obtained the metamodel for PL/SQL package declarations. The metamodel generated by Xtext is shown in the Figure 2. As can be seen, this metamodel is of poor quality because it mirrors the grammar structure, thus including some superfluous elements (e.g., `PackageSpec` metaclass). In this case, the metamodel was simple enough as not to be a problem. However, when dealing with larger grammars and metamodels, alternative solutions have to be looked into [21].

As we explained above, given models that conform to this metamodel, there are two alternatives to generate the desired wrappers: (1) using only model-to-text transformations and (2) combining model-to-model and model-to-text transformations. In this project we tried both approaches.

Wrapper generation: model-to-text approach. The model-to-text transformation was implemented in MOFScript since it has a simple syntax and it was a mature tool. The first version of the generator basically mapped each PL/SQL package to a Java class, each procedure to a Java method with no return type and each function to a Java method. There was also a piece of transformation devoted to the mapping of PL/SQL datatypes and Java primitive types and

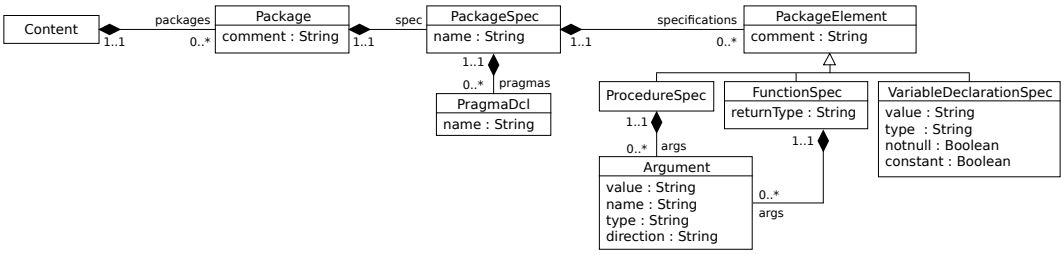


Figure 2: Meta-model generated by Xtext

library classes. The generator also performed a pre-processing of the comments in the PL/SQL code, so that they looked right in the Java wrapper.

This version was tested by the chief developer participating in the project, and five issues were reported through Trac. First of all, there were a modifier not considered in the PL/SQL grammar, but this was easily fixed. Secondly, the generated Java class required a specific naming convention. Thirdly, it was important to generate the contents of the Java class in a given order: constants, constructors and methods. Fourthly, overloaded PL/SQL procedures provoked conflicts in the generated code. Finally, import statements should not have used wildcards, but they had to comply with Java good practices and only the required classes be imported. The simple solution of parameterizing the imports through an external file was adopted.

The development of the third version of the generator required more complex modifications, since it was at this point when we were first reported on the existence of the company rules presented above. Notably, rules 6, 7, 8, 9 and 10 were implemented in this version. Most of the transformation rules required major modifications, and more transformation rules had to be included to deal with the specific functions of rules 6 and 7 (i.e., those ones beginning with `exists` and the `haveChildren` functions). In addition, a few bugs were fixed.

There was another version, the fourth one, that just implemented a few modifications, notably regarding naming and datatype selection. This was the final version that was deployed in Sinergia. Table 1 presents some figures that can give a hint about the complexity of the project increased as requirements were understood. Nevertheless, we will elaborate on the assessment of the implementation effort in Section 5.

Version	**Effort (Hours/Dev)**	**Rules**	**Helpers**	**Global var.**	**LOCs**
Version 1	+46	+7	+6	0	+334
Version 2	+33	+5	+3	+1	+76
Version 3	+70	+8	+1	+2	+249
Final version	+8	0	0	0	+40
Total	157	20	0	0	699

Table 1: Number of rules, helpers and global variables, as well as lines of code (LOC) for the versions of the project. The total effort involved in each version is also shown. The effort column shows the accumulated hours per developer dedicated to each version, which includes the time spent documenting the project.

An important aspect for the company was the deployment. Sinergia required that the generator was packaged so that it could be used out-of-the-box. To this end, we built a simple batch script that invoked ANT tasks to inject the given PL/SQL file and then apply the MOFScript transformation.

As the company guidelines were incorporated in the tool, the transformation became more complicated since the code that implemented them was scattered through several transformation rules. For instance, the transformation definition included three global variables, that were updated as the transformation rules were executed to detect function types (e.g., `exists` functions), zones, etc. It is important to clarify that these issues were not due to lack of care on the behalf of the developers, but it was inherent to using a MOFScript model-to-text transformation, where everything happens in a single pass. The transformation logic was too complex to be tackled with this strategy. However, this solution was successfully deployed in the company.

Wrapper generation: model-to-model approach. Even though the project was completed, we wanted to look into the extensibility and maintainability problems of the generator. We realized that part of the problem was that the company guidelines were hard-coded in the model-to-text transformation, and which is more, the transformation code that enforced the guidelines was tangled with the template code. In this way, we decided to re-implement the generator, by going through an intermediate model-to-model transformation written in RubyTL [22]. In a second step, the intermediate model was transformed into the Java source code by a model-to-text transformation.

With this experiment we wanted to assess if this approach could yield a more maintainable solution. The underlying idea was to explicitly represent in a meta-model the requirements regarding the different categories of PL-SQL code and the order of the different elements. Thus, a model-to-model transformation implements all the company guidelines, generating an intermediate model that makes the guidelines explicit for each PL/SQL package. We discuss the approach here, since it could be used in similar projects to improve its quality.

In this approach the intermediate metamodel, which is shown in Figure 3, is a key point. This metamodel satisfy the following objectives: (1) incorporating explicitly the company guidelines and constraints, such as the concept of zone (`Zone`) or constants (elements of `Constant` hierarchy); (2) simplifying the code generation by representing the structure of the wrappers, which become a simple traversal through the zones applying very simple templates; and (3) adding new elements in the future should be easy, issue which is facilitated by the use of a metamodeling language.

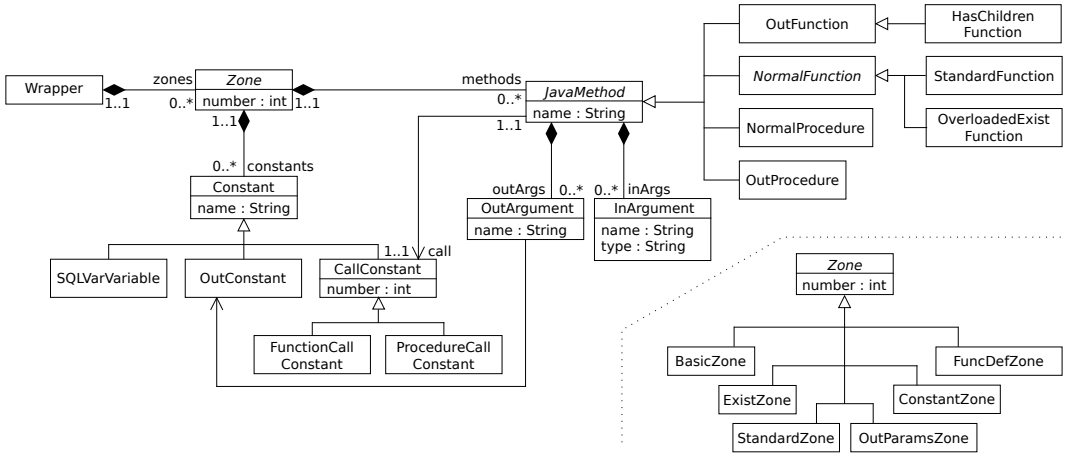


Figure 3: Meta-model to represent the generation requirements

Comparing this approach with applying a model-to-text transformation directly, an important advantage was that the logic to implement the detection of the different zones and functions was not scattered through the templates anymore, but encoded in clearly defined transformation rules and some navigation code. This also improved readability since most part of the model-to-text templates were plain text. Additionally, the intermediate metamodel served as documentation of the generation process, that otherwise would only be documented informally. Extensibility was also promoted since, if the company decided to incorporate a new rule related to zones, constants or functions/procedures, a new metamodel element would be added (probably inheriting from some existing metaclass), and the corresponding transformation rules would be written. Although three artefacts should be modified, the task is simple since it is easy to identify the places where modifications have to be made.

A potential problem of this approach is that the generator requirements have to be clear from the beginning, so that the intermediate metamodel can be built. In our case, for instance, this was not possible because we were not given the complete information until the second version of the generator was built.

4. Code generation project

4.1. Company details

The second project was carried out by Visualtis, a small company of 12 employees (at the time of the project) founded in Murcia in 2004. As they define themselves, Visualtis is "a leading edge software development company

with extensive experience designing and developing custom intranet software solutions for commercial and government organizations".

Visualtis usually works with open source software, either as software support (e.g., open source web frameworks) or by customising existing open source products such as ERPs or CRMs. They apply an agile methodology to the software development process, based on Scrum.

4.2. Project setup

This project was part of a transfer of MDE knowledge to three small regional software companies.The aim of this project, which was called AutoGSA (Automation with Generative Software Architecture), was to train the companies so that they were able design and implement a generative software architecture for a family of products. AutoGSA was launched and coordinated by the Regional Centre of ICT (Information and Communication Technologies), whose main objective is to promote the innovation in SMEs through ToT projects with universities and research institutes. The project was funded by the regional government and its duration was 14 months (it started at the beginning of 2009). It was organized in four phases: training, case study, pilot project and dissemination of results. Between two and three developers of each company received the training. They were software engineers, who often apply object-oriented programming (normally Java), and already had a background in UML modeling adquired as part of their academic degree.

Each company proposed then a pilot project to be developed during six months. Two researchers of our team visited each company in order to discuss the feasibility of the proposal and to achieve an agreement on the final requirements that should satisfy the product. At this point, the developers who had participated in the previous phases started to work in the pilot project, and three monitoring meetings and a final evaluation took place. In this paper we present the pilot project carried out by Visualtis since it was the only one successfully deployed. The other two companies did not comply with the project plan since they would not want to invest more time of their developers in the project. Instead they preferred to hire master students to develop the pilot project. Although we supervised both projects as well, unfortunately the products built were never put into practice within the companies (see Section 6.3).

4.3. Problem description

The project proposed by Visualtis was to automatize the development of the back-office for e-commerce and ERP applications. The company had already developed several systems of this kind, which were based on a clear component-based architecture supported by the Apache Tapestry framework [23] and Hibernate [24] as Object-Relational Mapping (ORM) framework. These systems usually involve the creation of numerous CRUD (Create-Read-Update-Delete) forms, which is a routine but time consuming and error prone task. Visualtis was aware that automating the development of CRUD forms would somehow improve its productivity. Hence, when they were requested to propose the pilot project, it was straightforward for them to come up with a potential generative software architecture [20].

As in the case of Sinergia, a clear approach based on manual coding already existed, and the project therefore consisted on its automation using MDE techniques. However, in this case it is a forward engineering problem. The intended generative architecture, called Circe, was based on a simple DSL to describe the business domain as object-oriented conceptual models, using classes, attributes and references as basic constructs. This DSL would be transformed into Java, XML and HTML code through transformations.

4.4. Example

To illustrate the problem we anticipate the presentation of the Circe DSL. Figure 4(a) shows the specification of a simple domain model to represent invoices. An `Invoice` is composed of `Items`, and has a `Customer` and a date. From this, two forms will be generated, one to manage customers (create, edit, delete operations), and another one to manage invoices. The generator must be smart enough to detect that items of an invoice must be managed as part of an invoice. Additionally, validators must also be created depending on the cardinalities of the properties. For instance, a validator must be created to ensure that the user enters the invoice date and that it has a proper format.

Figure 4(b) shows an excerpt of the Java class that should be generated for the `Invoice` business entity. As can be seen the class must be annotated with persistence and validation annotations. In addition, some care must be put on generating well formatted code, because as we will see it is likely to be modified.

```
package invoicedemo;

class Invoice {
    ref customer[1] : Customer;
    ref parts[*] container : Item;
    attr invoiceDate[1] : Date;

    -- Would be computed manually
    -- modifying the generated code
    attr total [1] : Float;
}

class Item {
    attr description[1] : String;
    attr quantity[1] : Integer;
    attr price[1] : Float;
}

class Customer {
    attr name[1] : String;
}

primitive String  = "String";
primitive Integer = "Integer";
primitive Float = "Float";
primitive Date = "java.util.Date";
```

(a)

```
package com.visualtis.invoicedemo.entities;

import javax.persistence.Entity;
// more imports...

@Entity
@SequenceGenerator(name="invoice_seq", sequenceName="invoice_seq")
@Table(name = "invoice")
public class Invoice {

  private Customer customer;
  private java.util.List<Item> parts = new java.util.ArrayList<Item>(0);
  private java.util.Date invoiceDate;
  private float total;
  private long id;

  public Invoice() {}

  public Invoice(Customer customer, java.util.List<Item> parts, java.util.Date invoiceDate, float total) {
    this.customer = customer;
    this.parts = parts;
    this.invoiceDdate = invoiceDdate;
    this.total = total;
  }

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "customer_customer_id")
  @Validate("required")
  public Customer getCustomer() {
    return customer;
  }

  @OneToMany(mappedBy="invoice", cascade = CascadeType.ALL)
  public java.util.List<Item> getParts() {
    return parts;
  }

  public float getTotal() {
    return total;
  }
  // more methods ...
}
```

(b)

Figure 4: Example of the Circe DSL. (a) Specification of simple Invoice model. (b) Generated code for the Invoice business entity.

4.5. Solution: the generative architecture

The devised generative architecture comprises basically three elements: an abstract syntax meta-model for the DSL, the specification of the concrete syntax and a series of model transformations. Next we comment on these elements.

The abstract syntax metamodel of the Circe DSL is shown in Figure 5. As can be seen it is simple and pragmatic since it just includes the features needed for the problem at hand, without additional complexity. For instance, it just has single inheritance since the target applications are generated in Java, and packages cannot be nested. This would facilitate subsequent model transformations.

The concrete syntax was developed with TCS [25], inspired by the KM3 meta-modeling language syntax [26]. The implementation of this component was relatively easy as well, since the developers had already practiced developing a similar concrete syntax with TCS (see Appendix A).

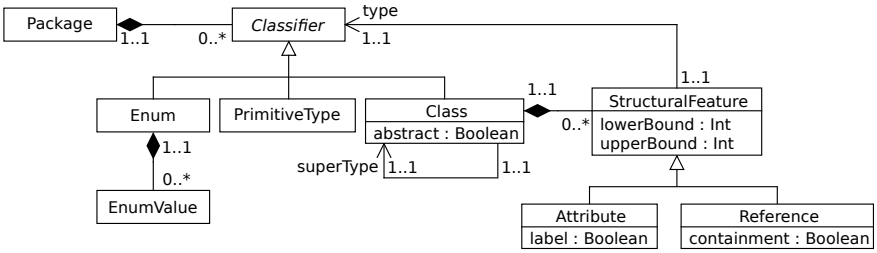


Figure 5: Abstract syntax metamodel of the DSL.

The aim of the project was to generate a usable, running application from a DSL program. Two well-known options are possible to achieve this goal in a generative architecture [20] [27]: creating a direct model-to-text transformation or splitting the task with intermediate model-to-model transformations. This decision was left to the company, so that they could choose the option they were more comfortable with. Interestingly, they decided to build a chain of model transformations. As reported by the company, the decision was driven by these three concerns.

- First of all, the intermediate metamodels represent the target platform explicitly, which facilitates reasoning about the generation process. Model-to-text transformations were in turn expected to be simpler.
- Some mappings between the DSL elements and the target platform were not straightforward. For instance, the well-known problem of converting class models (here the DSL program) to relational schema models required splitting the transformation into three phases to resolve circularities in foreign keys properly (they used techniques for modularizing transformations similar to those reported in [28]). Besides, transformation rules are more appropriate to recover some implicit information from the DSL, for instance different combinations of cardinalities and containment that lead to using different types of widget.
- Considering variants of the target platform, for instance, different database vendors. This issue was addressed by mark models that parameterized the model-to-model transformations.

The design of the generative architecture is depicted in Figure 6. A DSL program conforming to the `circe` metamodel is the input of three different model-to-model transformations. They are parameterized by mark models, which conforms to *circe_dbms_mark* or *circe_tapestry_mark* metamodels. These mark metamodels are intended to specify mappings between datatypes defined with the DSL and database datatypes and Tapestry widgets, respectively. So far, mappings for MySQL 5.1, Oracle 10g, Postgres 8.2 and Postgres 8.4, as well as for Tapestry 5.1 were defined.

The three intermediate metamodels that represent the target platform are shown in Figure 7. The first one, *Circe_ER*, is a typical metamodel to describe relational schemas, with the addition of `Sequence` metaclass to generate sequential identifiers for database tuples. (these sequences are used at the ORM level). The second metamodel, *Hibernate*, represents the domain entities mapped to the Hibernate ORM. The third metamodel, *Tapestry*, describes pieces of functionality that will be implemented using the Tapestry framework (represented as `Page` elements).

As can be observed the metamodels used within the project are not of great complexity, but they were developed by the company team (two developers) without any additional help, and which is more important, are enough for the

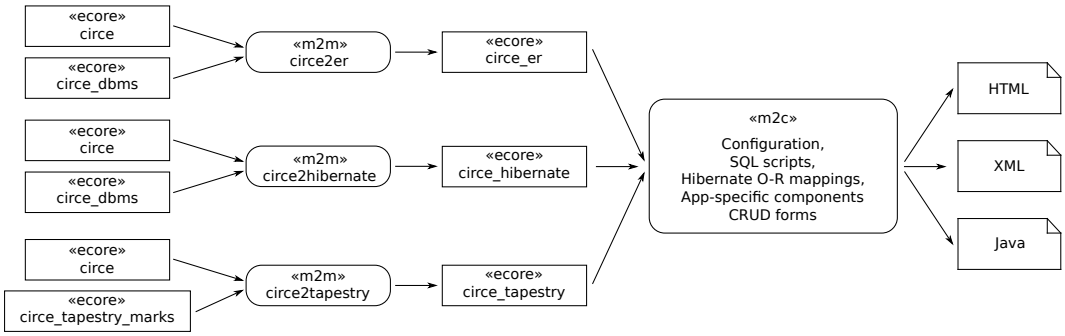


Figure 6: Design of the generative architecture implemented by Visualtis.

task at hand. For instance, it is worth noting that there are no explicit cross-references among models. Instead, names are used to refer to elements of a different metamodel created by another transformation. (e.g., a `Page` in the Tapestry metamodel refers to a Hibernate entity through the `entity` attribute).

The first version of Circe took 110 in total hours (around 55 hours per developer). It comprised almost all features expected for the final version. The second version took 40 hours in total, and added support for many-to-many relationships, a couple of changes in the user interface and the documentation of Circe. The changes to reach the final version took only 6 hours, and basically included support for a few datatypes (e.g., DateTime) and some facilities to make it easier execute the generator. Therefore, it can be roughly said that, with the proper training and knowledge of the target platform, a project of this kind can be completed by one developer in one month.

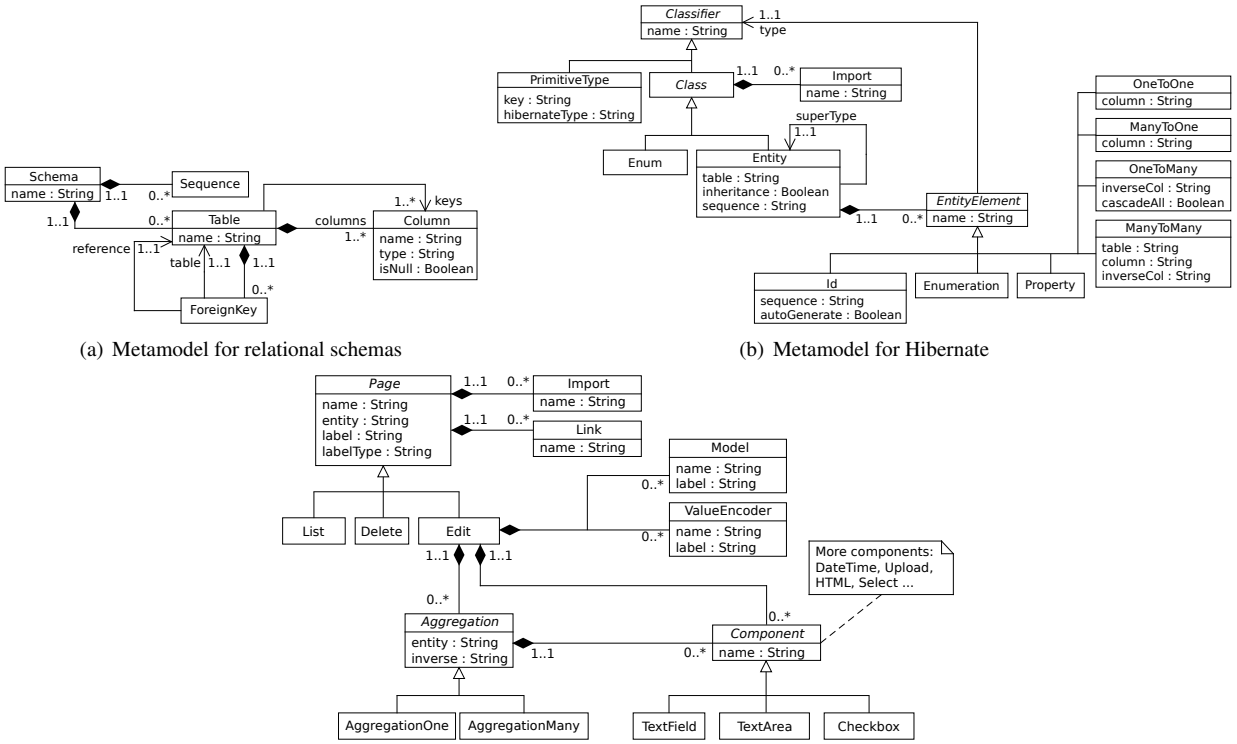
Regarding the number of lines of code involved in the project, Table 2 shows a summary (in Section 5 we will analyze the complexity of the project in more detail). First of all, metamodels were relatively small, in particular the abstract syntax of the Circe DSL. The concrete syntax was straightforward to implement. Most of the complexity of the project was in the model-to-model transformations, but no problems were reported about their development. The largest pieces of code correspond to the model-to-text transformations. However, they were implemented very rapidly because the input metamodels represented very explicitly what had to be generated. In this way, the transformation templates basically consisted of large pieces of text with a few holes filled with the information of the input model. In fact, the strategy to develop the templates was to take an existing project and copy-paste pieces of code to the templates.

Component	**LOCs**	**Comments**
Abstract syntax (Emfatic)	63	12 classes, 10 attr, 6 references
Concrete syntax (TCS)	248	204 lines of boilerplate code
Intermediate metamodels (Emfatic)	253	69 classes, 56 attr, 36 references
Circe2ER (M2M)	279	Written in RubyTL
Circe2Hibernate (M2M)	346	Written in RubyTL
Circe2Tapestry (M2M)	531	Written in RubyTL
Code generator	3428	Written in MofScript. Mostly copy-paste from an existing project.

Table 2: LOCs per component in Circe

An important advantage of the developed architecture was that the code generation step was greatly simplified. In fact, it was possible to split the task into 31 simple model-to-text templates, organized into five categories: configuration, SQL scripts, Hibernate ORMs, application-specific components and CRUD forms. In this way, fixing bugs in the generator was facilitated because each piece of the generated system was located in a well-known template.

It is worth noting that the developers were able of applying the knowledge acquired during the training on their own. For instance, in the theory part of the course they were taught about mark models to parameterize transformations and a simplified form of the class-to-relational transformation was used to illustrate model transformation languages. Besides, the experience gained during the practical part of the training made them capable of developing metamodels and writing model transformations.



(a) Metamodel for relational schemas

(b) Metamodel for Hibernate

(c) Metamodel for Tapestry

Figure 7: Metamodels that represent the target platform: a) relational schema, b) Hibernate ORM and c) Tapestry web component framework.

5. Assessment

In this section we assess the success of the considered projects according to the factors presented in Section 2. Table 3 summarizes them.

Factor	Modernization project	Generative architecture
Industrial context	Unfavorable	Unfavorable
Size of company	Around 100 employees	12 employees
Style of ToT project	University-centric	Company-centric
Kind of tool built	Small automation tool (reverse eng.)	Small automation tool (forward eng.)
Number of developers	2	2
Qualification of developers	Computer science (major)	Computer science (major)
Training (in MDE)	Informal	Formal (see Section Appendix A)
Previous work experience	None	A few years

Table 3: Summary of factors of the projects.

Next we comment on the industrial context and innovation, which was the same for both projects, and then we analyze the rest of the factors for each project.

Industrial context and innovation. Both projects described in this paper were developed in Murcia, the region in which our university is located. Murcia has an economy based on tourism and services, and whose per-inhabitant GDP in 2008 was 36% below the average of the European Union (UE-27). The performance of innovation of Spain is below the European Union (UE) average [29] and Murcia only contributes to the 1.56% of the Spanish investment in

technological innovation. With regard to the software development sector, Murcia has only SMEs in an approximate number of 250, from which an 80% have less than 10 employees and only three companies have between 150 and 200 employees [30]. It is worth noting that the R&D+i activities in small and medium software companies is extremely low in Spain, and, moreover, the adoption of MDE is lower than other European countries such as Germany, France or England. Thus, the industrial context regarding the ToT project described in this paper can be considered as unfavorable.

5.1. Assessment of the modernization project

In this section we will only consider the wrapper generator using the model-to-text approach, which was the tool used by the company.

Size and dynamics of the company. Sinergia has around 100 employees. Although it can be considered a small to medium company, due to its rigid structure it is mainly driven by inertia. This is not a surprise given the industrial context of our region and the fact that most of its business is devoted to projects for the regional government (a very conservative entity). Thus the limited degree of innovation of the company affected the expectations in the technology, that were low. This posed a problem because the company did not provide us with the complete set of guidelines to generate the Java wrappers from the very beginning. Although these guidelines were not formally documented, the chief developer participating in the project knew them. So it is interesting to wonder why the company did not provide us with these requirements at the beginning of the project. The reason can be found looking into the contents of the project management system, where those reports that notified us about the most complex guidelines (e.g., rules 6, 7 and 8) included sentences like the following (translated to English): *"We need something else that I do not know if it could be possible. Every function with no parameters and between* `exists` *and* `have[whatever]`*..."*. This denotes that at the beginning of the project, the expectations on MDE technology were limited, probably thinking that it could not be applied to tackle real problems.

Style of ToT project. The project was carried out in an university-centric style. The fact that this implied devoting few resources to the project probably encouraged the company to join it. At that time we saw this as an opportunity to show the company the potential of MDE, thus hoping that in the future they wanted to implement some pilot project by themselves. Unfortunately, while they really assessed the results positively, they still reckoned MDE as an arcane technology because they did not actively participated in the implementation.

Qualifications and training of the developers. The project was developed during three months by two part-time developers. They held a degree in computer science (5 years), which included subjects about object-oriented programming and UML modeling. They had learned about MDE informally because both of them had carried out a student project in which they had to learn and apply some MDE techniques. However, they lacked knowledge about important aspects such as model-to-model transformations.

Productivity gains. In order to assess the productivity gains obtained with this project we will analyze three issues: the effort required to implement the MDE tool, the effort required to use the MDE solution against the effort of using the manual approach, and the potential return of investment (ROI). To this end, as explained in Section 2, we apply some metrics [1].

To measure the *implementation effort* we analyze the effort required to develop the injector of PL/SQL models and then the generator, implemented in Xtext and MOFScript, respectively. We will take into account the development time (total hours, regardless of the number of developers) and we will use LOCs and some metrics about grammars, which will allow us to give a hint of the complexity of the languages used (Xtext and MOFScript).

Table 4 (second row) summarizes the results of the metrics about the implementation effort. Building the injector for PL/SQL was an easy task according to the low number of LOCs in Xtext and the value for the HAL metric applied to the PL/SQL grammar (that was generated by Xtext from the concrete syntax specification, but it gives a hint about the complexity of such an specification), which indicates that it is not a complicated grammar (i.e., designer's effort to understand the grammar is low). Moreover, the low values of LRS, LAT/LRS and SS metrics for Xtext also denotes

[1] The grammar metrics have been calculated with the tool "cfgmetrics", available for download at *http://code.google.com/p/cfgmetrics*.

that the language is easy to learn, thus facilitating the development. However, the implementation of the MOFScript generator was more costly. Although the implementation of the generator did not required a high number of LOCs, the implementation was not such an easy task, according to the high values of LAT/LRS and LRS metrics. On the other hand, the value of SS denotes that the language is quite verbose, which could help to improve the learning curve. Regarding the metamodel size, it could be considered small, and which is more important, it was automatically generated by Xtext. It is important to note that only PL/SQL procedure headers were needed in this project, thus simplifying both the grammar and the meta-model.

Case	Time	Grammar	HAL	Tool	LOC	LRS	LAT/LRS	SS	Metamodel
Modernization project	314 h.	PL/SQL	10.25	Xtext	98	454	0.13	1.87	8 classes, 19 attr., 5 ref. (generated by Xtext)
				MOFScript	699	7734	0.27	2.18	
Code Generation project	156 h.	Circe	4.1	TCS	248	18	0.13	1.61	12 classes, 10 attr., 6 ref. (abst. syntax) 69 classes, 56 attr, 36 ref. (int. met.)
				MOFScript	3428	7734	0.27	2.18	
				RubyTL	1156	14207	0.21	1.8	

Table 4: Implementation effort for the tool built in each project.

Next, we compare the effort involved in developing wrappers manually against using the automatic wrapping tool, which may serve as indicative of the productivity gain. Table 5 (second row) shows a summary for this project. The manual migration requires understanding the PL/SQL code in order to create the Java wrappers, which is a repetitive task following the recipe of the company. Thus, we have calculated some metrics for PL/SQL, which denote that the language is easy to understand (according to LRS values) and easy to learn (according to LAT/LRS and SS values). These values identify a simple and repetitive task but it is important to note that the migration requires creating a high number of wrappers, which is a tedious and error-prone task (as suggested by the LRS values for Java). In fact, the value of LRS for PL/SQL identifies it as a DSL (small, focused language to describe stored procedure signatures), so the task to generate Java wrappers (where LRS value is much higher) can be seen as a one-to-many transformation (a manual transformation initially) from a DSL to a GPL. On the other hand, the use of the tool does not require any understanding effort, since the tool acts as a black-box which automatically creates the Java wrappers, thus providing an important gain, in particular regarding to maintainability.

Case	Approach	Language	LRS	LAT/LRS	SS
Modernization project	Manual	PL/SQL	283	0.15	1.52
		Java	6244	0.18	2.04
	Generator	N/A	N/A	N/A	N/A
Code generation project	Manual	Java	6244	0.18	2.04
		HTML	6682	0.25	1.41
	Generator	Circe	102	0.09	2.8

Table 5: Use effort of the tool for each project.

With respect to the *return of investment*, according to the company's opinion, the deployed tool saved a considerable amount of time in comparison to developing the wrappers manually. It has been reported that a typical wrapper (i.e., wrapping all the functions of a PL/SQL package) took 3 hours on average to be developed by an experienced developed who already knew the company guidelines. Thus, taking into account the time involved in the creation of the generator (314 hours in total), 105 wrappers must be generated to recover the initial investment. It is worth noting that although the development of the generator requires a some effort, once available the development of a wrapper became a zero-effort task.

Project success assessment. Determining whether the project can be considered a success or not is done according to the dimensions presented in Section 2.

- *Knowledge dimension*. The company learned about the potential of MDE, but they got the impression that very specialized knowledge was needed.

- *Product dimension*. A well defined and properly scoped problem was choosen, and the final tool proved useful, so showing the applicability of MDE.
- *Adoption dimension*. The success of the project encouraged the company to try to apply the MDE technology in similar contexts. For instance, as a continuation of the same project they asked us to support the generation of web-services to access PL/SQL packages. However, the company was not really involved with MDE, but they depended on us to apply the technology. Later, the company participated in a company-centric ToT project but they did not invest actual developer time in building a tool for the company, so this effort had no impact on the company development process.

In summary, it could be considered as a partial success, as the main goal of the project was to demonstrate MDE by building a useful tool, although it is clear that we were unable to put the company on the way of adopting MDE to build automation tools.

5.2. Assessment of the code generation project

As done in the previous section, the third column of Table 3 shows a summary of the factors involved in this project. Next, we elaborate on them.

Size and dynamics of the company. Visualtis has around 12 employees. It is a small company, with a dynamic profile that, in general, it is open to try new technologies.

An important aspect for the company to apply Circe in practice was whether it fits well in its development process. They apply an agile methodology based on Scrum, which is inherently iterative. They found out that Circe was a good complement for the life cycle, because it allowed them to specify the domain model very easily at the beginning of an iteration, and to obtain an implementation automatically, that will be modified during the iteration. The only concern was to keep manual changes after re-generation. This was addressed by a technique in which the Git [31] control version system was used as a way to detect the manual changes of the code after regeneration and perform a seamless merging of the new, generated code and the manually modified one.

This experience is an example that the attitude of the company and its developers towards new technologies is a key element to overcome problems (such as incremental consistency) that could jeopardize the success of the project. On the contrary, having solved this problem Circe became part of the company's usual development process.

Style of ToT project. The project was carried out in a company-centric style. As explained, two developers of the company received specialized training in MDE (more details in Appendix A), and then they applied this knowledge to the development of the pilot project, the Circe DSL.

Qualifications and training of the developers. The two developers involved in the project held a degree in computer science (5 years), which included subjects about object-oriented programming and UML modeling. They did not know about MDE, but they were trained as explained in Appendix A. They had around four years of experience as professional developers, particularly focused on object-oriented programming and web development frameworks.

Productivity gains. We have assessed the productivity gains in this project in a similar manner as with the previous project.

The generative architecture required the implementation of a DSL using TCS and implementing some model transformations using RubyTL and MOFScript. Moreover, four metamodels were also developed: the abstract syntax metamodel and three intermediate metamodels.

Table 4 (third row) summarizes the results of the metrics we have applied to measure the *implementation effort*. According to the figures, the task of building the concrete syntax and the abstract syntax metamodel was simple and straightforward. First of all, the meta-model was small. Regarding the concrete syntax, the low value of HAL for Circe indicates that its underlying grammar is easy to understand, and which is more the LRS and LAT/LRS metrics indicate the simplicity of TCS (i.e., a DSL to create the concrete syntax of another DSL). Most of the complexity of the project was in the model-to-model transformations. As can be seen, the model transformation languages used in this project (i.e., RubyTL and MOFscript) are relatively complex languages. As indicated in the previous project, MOFScript is not straightforward to use. RubyTL is in this sense similar to MOFScript, according to the LRS and

LAT/LRS metrics, and the low value of SS could denote that the language is easy to learn. However, it is worth noting that both model transformations were implemented without any remarkable problem, because the input metamodels represented very explicitly what had to be generated. In this way, the transformation templates basically consisted of large pieces of text with a few holes filled with the information of the input model. In fact, the strategy to develop the templates was to take an existing project and copy-paste pieces of code to the templates.

Comparing the manual approach against the use of the DSL could help us to give some grounds about productivity gains. As explained in Section 4, a manual approach requires working with Java and HTML code. Table 5 (third row) contrasts the languages used in each alternative. As can be seen, the values of the metrics clearly denote that the use of HTML/Java requires much more effort than the use of the DSL developed within the project.

From a practical point of view, a Circe program with five classes, five attributes and five references generates around 3000 lines of Java code, 1000 lines of HTML files, and a few configuration files. A medium-size application typically involves around 25 domain classes with between 5 and 10 features each. Thus, it is clear that with the manual approach a great amount of boilerplate code was written, while with Circe the developers only have to concentrate on the domain model and the business logic.

Regarding the ROI, we do not have precise data to compare the time spent originally using the manual approach against the use of Circe. However, according to the productivity data reported by Visualtis, the development time for applications that require CRUD forms is shortened from 70% to 90%. If GUI personalization is needed then the shortening is between 60% and 70% since part of the task is manually performed by modifying generated code.

The generative architecture created by Visualtis was successfully applied in several e-commerce application projects. In particular, as reported by the company, it was used for five projects from 2010 and 2011. Circe also proved to be useful in rapid prototyping. The company reported on a project related to mobile devices and geo-positioning, where they needed to generate services and mobile applications. The first version of the administration forms to show the idea to the client was generated with Circe, and later refurbished specifically for mobile devices.

Thus, it can be said the Visualtis was able to create an MDE-based automation tool and to integrate it into its development process. As a matter of fact, when the results of the AutoGSA project were publicly presented, the Visualtis manager expressed *"... but the most important thing for us has been to acquire knowledge and skills in MDE as if in the future we change the development frameworks, we will be able to create a new generative architecture"*. These facts show that the ToT project could be, in principle, considered as successful.

Project success assessment. The following assesses the success of the project according to the dimensions presented in Section 2.

- *Knowledge dimension*. The company not only learned about the potential of MDE, but they learned how to apply MDE in practice. Which is more, they even devised a custom solution for the problem of incremental consistency of the generated code.
- *Product dimension*. A well defined and properly scoped problem was choosen, and the final tool proved useful, and was integrated in the development process of the company.
- *Adoption dimension*. From this project the company, when faced to a transformation problem or automation problem, was open to consider MDE as a candidate technology. We have been reported that they wanted to apply MDE to tackle some problem in an industrial project, but they decided to discard it because they felt that the MDE tools were immature for a non in-house project. In fact, a couple of years after this project, one of the developers of Circe moved to another company where he introduced MDE by building a DSL to automate a task that was performed manually (now with more mature tools: Xtext, ATL and Acceleo).

In summary, this project could be considered as a success in every dimension, although the immaturity of the MDE tools at that time perhaps stopped a broader adoption.

6. Lessons learned

In this section we will present some lessons we have learned from the experience gained during the development of these projects. They may be useful both for small companies that want to apply MDE and research teams leading ToT projects.

6.1. Technical issues

Although the tools built in each project are completely different (i.e., a reengineering tool and a generative architecture), it is interesting to identify some commonalities and differences.

In both projects model-to-model transformations were essential to obtain intermediate models which describe effectively the target architecture and facilitate the final model-to-text transformation. The metamodels involved in the processes were therefore a key aspect, whose definition, from our experience, requires high-prepared engineers with abstraction capabilities.

The mapping of datatypes between languages was also a common issue in both projects, in this case from PL/SQL to Java, and from the Circe DSL to SQL and Java. This can be implemented within the transformation definition, but using an external model (i.e., a mark model) that parameterizes the transformation is a more flexible approach.

When dealing with Java code generating import statements following good practices seems to be important for the companies. Precise generation of import statements was actually a concern in the code generation step of both projects. There is no general solution for this problem. In the first project an external file was used to parameterize the transformation, while in the second one the Tapestry metamodel was created with a built-in `Import` metaclass and the transformation definition was in charge of creating the imports to the concrete classes of Tapestry.

The evolution of the generated code has to be addressed to achieve a successful product and this is sometimes considered as one of the weak points in the adoption of MDE in industry. The experience of Visualtis has been really successful in this aspect, since the developers were able to implement a solution for the incremental consistency issue that is integrated into the Scrum process and tools used in the company.

6.2. Choice of the pilot project

We have found that the staff of small software companies is really impressed with the possibility of automating software tasks and automatically generate code from models, although it seems that some of them are very skeptical about its scalability at the same time. It therefore turns out that the choice of the right pilot project is crucial for the success of the ToT project.

For small companies, small projects that automate repetitive tasks in a simple way are a good manner to take advantage of the possibilities of MDE. This is so, because MDE provides a systematic way of building DSLs and automation tools, and more importantly, building them does not require spending a large amount of developing time and produce useful and visible results in a short time. To a large extent, this is so thanks to the existence of compatible tooling and languages, notably the projects built around the EMF meta-modeling framework. Both projects presented here are examples of projects appropriately scaled to the resources available in small companies.

6.3. Company size

The description of the projects shows that the commitment of the companies with the projects was different. In our experience, the smaller the company the easier to get a real involvement in the project. This is so because small companies have more flexibility and respond better to the changes as indicated in [8]. As the size increases the organizational inertia grows. Richard Daft has noted that the flat structure and management style of small companies encourages entrepreneurship and innovation [32]. Our experience bears out this observation as Visualtis (12 employees) finished successfully the pilot project and they have incorporated MDE to its development process, while Sinergia (100 employees) has shown some reluctance to apply MDE techniques despite they have evidences of the benefits of this technology.

The agile methods are increasingly used in small companies as they are recommended to teams of around 5 to 15 developers. The Circe project is an experience showing that a MDE tool, in particular a generative architecture, can be integrated as part of the iterative and incremental development promoted by an agile method (i.e. Scrum).

6.4. Involvement in the project

Two different styles to transfer technological knowledge to companies have been compared: university-centric and company-centric. In both cases, useful tools that automate repetitive and error prone tasks were built, and the companies adquired knowledge about MDE. However, the involvement of the companies was different.

In the first project the implementation work was performed by two engineers hired by the research team, which was enough to obtain the desired product, deploy it in the company, and use it effectively. However, it was not

enough to engage Sinergia with MDE, because we provided a solution packaged out-of-the-box, so that the concrete techniques used to implement the solution became irrelevant for them.

Our hope at the beginning of the project was that building a useful tool would make the company interested in the underlying techniques. Unfortunately that was not the case, even though we presented the results to the company managers at the end of the project trying to explain that we had used MDE techniques and the possibilities that MDE brings.

On the contrary, in the project with Visualtis, the company carried out the development of the tool by itself. Besides, they had already invested some time in the training stage. In our experience, the critical point happens after the training phase, where the company managers must decide whether to start with the pilot project to apply the knowledge acquired in the training phase. The smaller the company, the higher the likelihood that the management team agrees with investing more developer time. In this sense, it is important to keep the training phase short and the pilot project small and focused to automate a real task of the company.

Thus, we have witnessed that the second approach (company-centric) is more appropriate due to two reasons. First of all, involving company developers in an implementation project makes them realize the productivity gains that MDE may bring. Secondly, synergies come up between industry and academia that may lead to new ideas and developments. This is clearly illustrated by the Git-based solution to incremental consistency devised by Visualtis.

Regarding the training phase, we learned that it is important to keep it focused and lightweight, trying not to load trainees with additional extra work that is not of direct applications to their daily work. More details are given in Appendix A.

6.5. Project evolution

Finally, an important concern of any MDE project is its evolution as new requirements, new frameworks or new tools frequently appear. In this sense, Visualtis has reported that they have problems with the integration of Circe in its new build chain based on Maven because the execution of the generator tasks was done with some predefined ANT tasks. They also would like to have an Eclipse plugin that integrates the generator seamlessly, instead of creating a new ANT build.xml file for each different project. In addition, we have been reported that when facing a problem of translation between data formats they considered using MDE techniques, but they were hold back because of the immaturity of the tools. If we had been closer to the company after the project, we could have advised them about the evolution of some tools since they were trained.

In the case of Sinergia, they stopped using the wrapper generator when the guidelines were changed. The generator built as part of the project was used to wrap around a hundred PL/SQL packages, so the return of investment in the project was limited. Probably small changes in the generator would have allowed them to keep using it, but nobody in the company had the knowledge to make the generator evolve.

In retrospective, we failed in following the evolution of both projects once the funded ToT projects were finished, for instance setting up new projects or just by arranging a meeting. In some sense, we neglected the adoption phase. Therefore, it is important to keep track of the usage of the results of the ToT projects in the companies, since the research team may give useful advice with little effort, that will probably lead to a better return of investment in MDE technology.

7. Related work

The number of experiences describing how MDE is applied in industry is still limited, and they are normally case studies or pilot projects carried out in large companies that have decided to adopt MDE (or at least have tried). A review of such experiences has been performed by Mohagheghi [33]. He organizes the evidences and conclusions around three questions: where and why is MDE applied?, what is the state of maturity of MDE?, and what evidence do we have on the impact of MDE on the productivity and software quality? The review analyzed 25 papers and revealed that: i) the majority of them does not provide information to estimate the size of the project; ii) 20 of papers were experience reports, from which only two included quantitative data on the projects and only 7 were completed projects; iii) 3 of papers were comparative studies, and from these, only one provided quantitative data. Other examples developed in large companies are [34] and [35], where the experiences within Motorola and Saturn car manufacturer are presented, respectively. In both papers, authors discuss about the impact of the paradigm in the

organization, the issues encountered and lessons learned throughout the development of the projects. This paper is an experience report on two projects aimed at transfering MDE technology to small companies. The projects were completed and the tools built have proven to be useful for the companies involved. We have provided information on the size and nature of the problems addressed, as well as quantitative data, including some metrics.

Recently, some results of an empirical study on MDE, based on questionnaires and interviews, have been published by Hutchinson et al. [9]. The aim of this research has been to understand and document how MDE is being applied in the industry and identify the factors affecting the success or failure of MDE. In the projects presented here, we have had to face some of the issues raised in the responses to questions of the study. For instance, the need of keeping models and code synchronized is noted as a critical issue for the success of MDE. This issue was an essential aspect of the generative architecture implemented by Visualtis. With regard to training, a 74% of respondents consider that MDE requires them to carry out a significant extra training, and some interviewees have claimed that adopting MDE could have a significant risk as it could be difficult to have developers with an MDE education. In our company-centric project we set up a MDE training phase to address this issue, which is also reported in Appendix A.

Hutchinson et al. have also presented another empirical study describing the MDE practices at three large companies [6]. The work highlights the importance of several organizational aspects: a commitment of the organization, a progressive and iterative adoption, a motivation of developers in applying MDE, the adaption of the existing development processes, and an adoption focused to new challenges. We have expressed similar lessons learned for small companies by contrasting the results of Sinergia and Visualtis. An interesting final consideration of the study is that MDE is essentially a technical vision of the software development and therefore companies adopting MDE have to identify which MDE techniques affect their development process beyond organizational changes. The discussion on the usefulness of model-to-model transformations and the incremental consistency approach devised by Visualtis are examples of such issues.

Another empirical study based on questionnaires and interviews has presented results on the MDE adoption at two large companies [36]. It gives a set of conditions to be satisfied by the companies to increase the chances of succeeding in adopting MDE: maturity of technology, compatibility of MDE with the software process used in the company, staff available in the company with the required expertise, and the precise definition of goals for introducing MDE. We addressed the training of company staff and we have commented on the compatibility of MDE with the Scrum process followed in Visualtis.

A study on the adoption of MDE in a small-medium company was performed by Vogel and Mantell [37]. However, they do not analyze the implications of the size of the company in adopting the technology, but they only discuss the approach applied by the company and the initial results obtained. The authors briefly present the education plan developed to train in MDA. They do not distinguish between theory and practice. Besides, their aim is to change the company code-centric approach to a model-centric approach. In our case, we claim that introducing MDE should start with small projects that align with the company development style.

8. Conclusions

The technology transfer from academia to industry is a key action to achieve a widespread adoption of MDE. Most reports on MDE adoption in industry involve only large companies, but small software companies represent about 90% of all software enterprises. Small companies face similar software engineering problems as large ones, but their characteristics are different so the solutions should be different as well. Thus, the introduction of MDE in a small company must take this issue into account, particularly when scaling the initial MDE projects of the company.

In this paper we have reported on our experience developing two ToT projects. The projects involved two different software companies, a small-medium company of around 100 employees and a small company of 12 employees, and had the goal of engaging the companies with MDE techniques. Both projects were successful from the knowledge and product dimensions, as useful tools were built that automated some software development tasks and the companies learned (in different degree about) about MDE. However, from the adoption dimension the results were disparate: in one of the projects we failed to engage the company with MDE, while in the other at least the company adopted MDE as a candidate technology to build automation tools.

It is our hope that the experiences described here are useful both to enterprise and academia. For the enterprise, both case studies are examples of MDE projects that do not require a large amount of resources but do improve

productivity. They can be used as a starting point to devise others projects that help the enterprise automate some repetitive task. The description and evaluation of both projects illustrates the cost of building this kind of solutions, and gives some insight about when it is worth the investment. For the academia, being successful in a ToT project is really difficult due to the inherent nature of most software companies. The lessons learned that we have commented could be of interest to other research teams carrying out some ToT project. In this respect, we contribute our approach to training in MDE in an appendix. The material of the course is also contributed, so that it can be either directly used or adapted.

Acknowledgements. We are grateful to Pedro Luis López Sánchez and Carlos Castillo for their rapid and precise responses about the projects. We also credit Pedro for devising, and sharing with us, the incremental consistency mechanism based on Git. This work has been supported by the Spanish Ministry of Economy and Competitivity (project "Go Lite" TIN2011-24139), by the European Commission under the ICT Policy Support Programme, grant no. 317859, and by Fundación Séneca-CARM (grant 15389/PI/10).

References

[1] OMG, MDA Guide Version 1.0.1, http://www.omg.org/mda (2003).
[2] S. Kelly, J.-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE CS, 2008.
[3] A. Clark, P. Sammut, J. Willans, Applied Metamodelling: A Foundation for Language Driven Development. Second Edition, Ceteva, 2008, see also `http://itcentre.tvu.ac.uk/~clark/Publications.html`.
[4] OMG, Architecture-Driven Modernization (ADM), http://adm.omg.org/ (2007).
[5] B. Selic, Personal reflections on automation, programming culture, and model-based software engineering, Automated Software Engg. 15 (2008) 379–391.
[6] J. Hutchinson, M. Rouncefield, J. Whittle, Model-driven engineering practices in industry, Software Engineering, International Conference on (2011) 633–642.
[7] B. Selic, What will it take? a view on adoption of model-based methods in practice, Software and System Modeling 11 (2012) 513–526. doi:10.1007/s10270-012-0261-0.
[8] I. Richardson, C. G. von Wangenheim, Guest editors introduction: Why are small software organizations different?, IEEE Software 24 (1) (2007) 18–22. doi:http://doi.ieeecomputersociety.org/10.1109/MS.2007.12.
[9] J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, Empirical assessment of mde in industry, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 471–480. doi:10.1145/1985793.1985858.
URL `http://doi.acm.org/10.1145/1985793.1985858`
[10] L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, T. Yue, Research-based innovation: A tale of three projects in model-driven engineering, in: C. A. R. B. Robert France, Juergen Kazmeier (Ed.), Model Driven Engineering Languages and Systems, 15th International Conference, MODELS 2012, Springer, 2012.
[11] B. Selic, What will it take? A view on adoption of model-based methods in practice, Software & Systems Modeling 11 (4) (2012) 513–526.
[12] J. Aranda, D. Damian, A. Borici, Transition to Model-Driven Engineering What Is Revolutionary, What Remains the Same?, in: MODELS conf., Vol. 7590, LNCS, 2012, pp. 692–708.
[13] T. Clark, P.-A. Muller, Exploiting model driven technology: a tale of two startups, Software and System Modeling 11 (2012) 481–493. doi:10.1007/s10270-012-0260-1.
URL `http://dx.doi.org/10.1007/s10270-012-0260-1`
[14] E. European Comission, Industry, The new sme definition. user guide and model declaration (2012).
[15] M. Crepinsek, T. Kosar, M. Mernik, J. Cervelle, R. Forax, G. Roussel, On automata and language based grammar metrics, Computer Science and Information Systems 7 (2) (2010) 309–329.
[16] E. Software, The Trac Project, http://trac.edgewall.org/ (2012).
[17] R. C. Seacord, D. Plakosh, G. A. Lewis, Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
[18] E. J. Chikofsky, J. H. Cross, Reverse engineering and design recovery: A taxonomy, IEEE Software 7 (1) (1990) 13–17.
[19] J. Canovas, J. Molina, An architecture-driven modernization tool for calculating metrics, IEEE Softw. 27 (2010) 37–43. doi:http://dx.doi.org/10.1109/MS.2010.61.
URL `http://dx.doi.org/10.1109/MS.2010.61`
[20] T. Stahl, M. Voelter, K. Czarnecki, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006.
[21] J. L. Cánovas Izquierdo, J. G. Molina, A domain specific language for extracting models in software modernization, in: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 82–97.
[22] J. Sánchez, J. García, M. Menárguez, RubyTL: A practical, extensible transformation language, in: ECMDA-FA'06, Vol. 4066 of LNCS, Springer, 2006, pp. 158–172.
[23] Apache Tapestry, http://tapestry.apache.org/ (2012).
[24] Hibernate ORM, http://hibernate.org (2012).
[25] F. Jouault, J. Bézivin, I. Kurtev, Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering, in: Proceedings of the 5th international conference on Generative programming and component engineering, GPCE '06, ACM, New York, NY, USA, 2006, pp. 249–254.

[26] F. Jouault, J. Bézivin, KM3: A DSL for metamodel specification, in: FMOODS'06, Vol. 4037 of LNCS, Springer, 2006, pp. 171–185.
[27] S. J. Mellor, K. Scott, A. Uhl, D. Weise, MDA Distilled, Addison-Wesley Object Technology Series, 2004.
[28] J. S. Cuadrado, J. G. Molina, Modularization of model transformations through a phasing mechanism, Software and System Modeling 8 (3) (2009) 325–345.
[29] I. U. ScoreBoard, The Innovation Union's performance scoreboard for Research and Innovation, http://ec.europa.eu/enterprise/policies/innovation/files/ius-2011_en.pdf (2012).
[30] N. S. I. of Spain, Number of companies by industrial sector, http://www.ine.es/ (2009).
[31] G. DVCS, Git - Fast version control system , http://git-scm.com/ (2012).
[32] R. L. Daft, Organisation Theory and Design, West Publishing, 1992.
[33] P. Mohagheghi, V. Dehlen, Where is the proof? - a review of experiences from applying mde in industry, in: I. Schieferdecker, A. Hartman (Eds.), Model Driven Architecture Foundations and Applications, Vol. 5095 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 432–443.
[34] P. Baker, S. Loh, W. Franck, Model-Driven Engineering in a Large Industrial Context Motorola Case Study, in: MODELS conf., 2005, pp. 476–491.
[35] E. Long, A. Misra, J. Sztipanovits, Increasing productivity at Saturn, Computer (August) (1998) 35–43.
[36] M. Staron, Adopting model driven software development in industry a case study at two companies, in: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (Eds.), Model Driven Engineering Languages and Systems, Vol. 4199 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 57–72.
[37] R. Vogel, K. Mantell, Mda adoption for a sme: Evolution, not revolution - phase ii, in: Workshop From Code Centric to Model Centric (ECMDA 2005), 2005, iNT LIP6 MoVe.
[38] Training in MDE material, http://mde-training.com/ (2012).
[39] J. Bézivin, F. Jouault, Using atl for checking models, Electron. Notes Theor. Comput. Sci. 152 (2006) 69–81.

Appendix A. Training

In this section we summarize our approach to teach MDE. This was put into practice in the AutoGSA project, to which the pilot project carried out by Visualtis belongs to. Besides, it is the basis of a Master's course in MDE at the University of Murcia. As will be shown, we tried to keep the course short and focused, since companies are not typically willing to spend a lot of time in training. Also, we tried to make a balance between theory and practice. The material of this course is available for others to use or adapt in [38].

In this way, the training is split into theory and practice. Theory is taught in a classroom during 3 lectures of 4 hours each. The goal is to present the main concepts underlying MDE so that they can be applied later in the practice sessions. A variety of concrete examples that show the applicability of MDE in industry are also introduced. Even though theory is based on a series of lectures, we try to teach MDE in a practical way. Previous knowledge of object-oriented programming and UML modeling is assumed, which is reasonable since it is part of most university curricula since some years ago. The organization of this part is as follows:

- Introduction to Model-Driven Engineering (motivation, basic principles and different visions)
- Meta-modeling (basic concepts, first examples, metamodeling languages, MOF and EMF architectures, and OCL)
- Building DSLs (elements of a DSL, different approaches, and tools for defining DSLs)
- Model transformation (model-to-model transformations and model-to-text transformations)
- Practical applications and examples

At the end of this stage, the trainee has learned the basic concepts of the paradigm and knows the characteristics of different MDE languages and tools. The next step therefore would be to practice these concepts using some selected tools and languages.

Appendix A.1. Practical sessions

In this section we summarize how we organize the practical sessions. We believe that this is of interest because currently there is a lack of comprehensive, practical courses in MDE.

The practical part of the training consists of six sessions of three hours each, where concrete tools and languages are used to solve a running case study. Our aim was to design a case study with the following characteristics:

```
application menu EclipseMenu;

tree menu File {

  icon "/ file .jpg"
  shortcut "Alt+F"
  mnemonic "F"
  tooltip "File"

  action menu New {
     mnemonic "N"
  }

  action menu Close {
     shortcut "Ctrl+W"
     mnemonic "C"
     tooltip "Close"
  }

  checkbox menu Synchronize default true {
     shortcut "Ctrl+S"
     mnemonic "S"
     tooltip "Auto Synchronize"
  }

}

tree menu Edit {
  icon "/ edit .jpg"
  shortcut "Alt+E"
  mnemonic "E"
  tooltip "Edit"
  action menu Undo { ... }
  action menu Redo { ... }
}

tree menu Help {
    ...
}

tree menu Install {
    ...
}

profile NoviceUser {
  use File : block (Synchronize)
  use Edit : hide (Undo, Redo)
  use Search
}

profile AdvancedUser extends NoviceUser {
  use Window : merge (Install)
}
```

Figure A.8: Example of the concrete syntax of the DSL

- It should be moderately complex, so that it allows different tools and languages under study to be used. It is also important that it can be solved incrementally.
- It should tackle a practical problem. It is not expected that the results are of direct application but it should show how to solve similar problems, adapted to the company needs.
- Automating a task by creating a DSL is a good exercise, because it may involve meta-modeling, concrete syntax (textual and graphical), model-to-model transformations and text-to-model transformations. Additionally, it requires abstracting from the generated product.

The case study we devised was a DSL to automate the creation of menus of desktop or web applications independently of the target platform. It was created as a result of some conversations with a developer about typical repetitive tasks carried out by software companies. The DSL has to allow defining different types of menu options: tree menu, a menu that links an existing menu, a single menu element that perform an action or that is a checkbox. It has to permit defining menu profiles, that is, a set of menu options specifically designed for a certain kind of user. However, it is expected that profiles share certain menus, so an inheritance mechanism for profiles must be implemented, with three operations to compose menus: block, hide or merge some inherited menu tree. Finally, the internationalization aspect has also to be considered.

Figure A.8 shows a specification of menus and menu profiles with this DSL, and Figure A.9 shows the abstract syntax metamodel of the DSL. As can be seen, implementing the abstract syntax metamodel is not straightforward, but it can be done iteratively, first addressing the menu hierarchy, then considering profiles and finally menu inheritance. Besides, it allows interesting transformation problems to be explained and practiced, for instance, resolving linked menus or dealing with operations to compose menus.

In general, in each practical session the corresponding part of the case study is presented and trainees are given some time to implement a solution. During this time the trainer supervise the trainees' progress and help them if needed. Afterwards, a canonical solution is presented, and commented with respect to the obtained solutions. Nevertheless, the case study is designed so that little variation is normally possible, because it is important to have a common baseline to address the following exercises. Next, we explain the different practical sessions.

Meta-modeling. This part comprises two sessions, where the basic concepts of meta-modeling are put into practice by building some metamodels with EMF and EMFatic (a textual syntax to specify EMF metamodels). XMI is also introduced by examining the models and metamodels generated by EMF. Before actually starting the session, the case study is presented and a brief introduction to Eclipse is given if the trainees do not have previous experience.

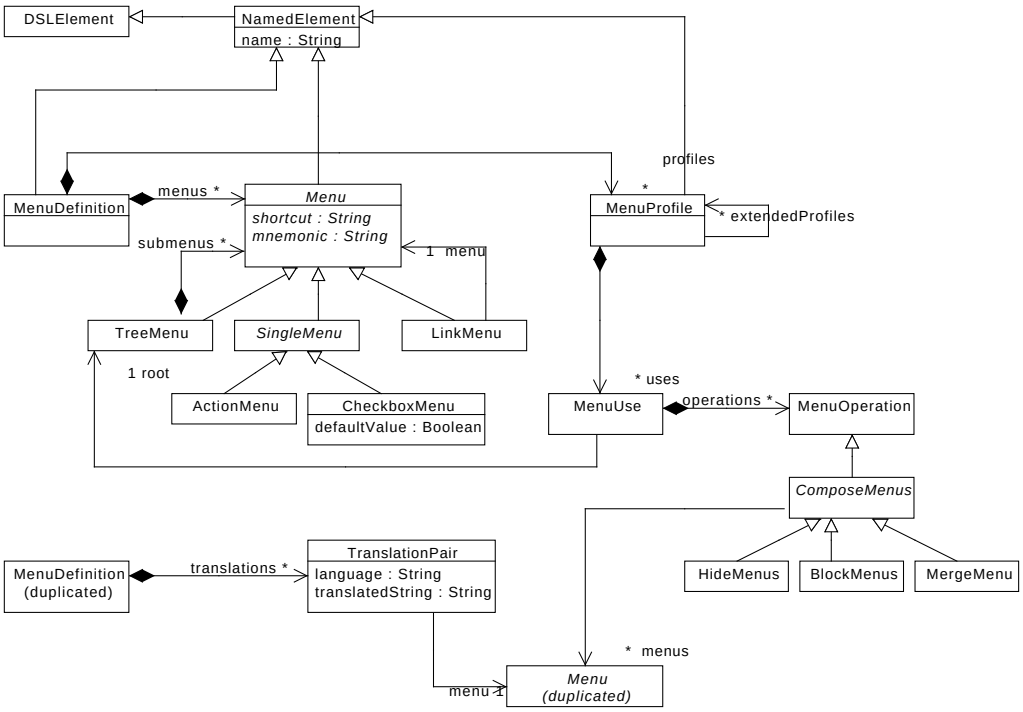


Figure A.9: Meta-model of the abstract syntax of the DSL to represent menus.

The first part of the second session is guided. The trainees are given a diagram with an excerpt of the abstract syntax of the DSL that will be built in the case study. This metamodel is discussed with them, and then they proceed to implement it using EMFatic, as well as creating some example models using EMF facilities.

In the second part they have to complete the initial metamodel to include internationalization features and reuse mechanisms as explained above. Afterwards, they are given a canonical solution, and the obtained metamodels are commented with respect to them. Finally, they have to build a metamodel to represent the code that will be generated from the DSL (i.e., the task basically consists of meta-modeling the usage of the Swing API for menus). Again, the results are commented.

Textual concrete syntax. This session aims at showing how to give a textual concrete syntax to a DSL, making special emphasis on how to map identifier-based references of textual definitions into explicit references in the abstract syntax model. Some examples are first introduced and then it is explained how the syntax for the DSL of the case study must be defined. This session is more straightforward than the previous one, since it concentrates on the basics leaving aside more complex issues such as translating expressions. The trainees have to implement the concrete syntax of the metamodel created in the previous session. Initially this session was prepared for TCS, but it has been updated to use EMFText and Xtext.

Model-to-model transformations. This part comprises two sessions, since learning a concrete model transformation language is in our experience one of the most difficult elements of MDE. As implementation language RubyTL was chosen, but it can be easily adapted to other languages, in particular ATL or ETL. In addition, in this session model validation is also introduced, by using a model-to-model approach as described in [39].

In the first session, the RubyTL model-to-model transformation language is introduced by implementing in a guided fashion a simple example where a UML class diagram is transformed into a Java model. Then, the trainees will be requested to perform slight modifications on the transformation definition, in order to observe the effect in the

target model. Afterwards, they are given a new exercise that have to implement on their own during one hour. This exercise intends to reinforce the knowledge about basic transformation rule behaviour and model navigation.

In the second session they have to implement the model-to-model transformation between the DSL and the target architecture metamodel created in the meta-modeling session. To start with, they have to implement a series of validation rules for the DSL. Then, they implement the actual transformation incrementally, with the trainer checking that the different steps are properly achieved. The steps are: (1) basic implementation without considering neither inheritance nor use of attributes such as tooltip, (2) consider basic profile inheritance, (3) consider attributes such as tooltip, (4) avoid transforming menus marked as *hidden*, (5) consider *block* menu operations and (6) consider *merging* menus, and (7) consider redefinition when a profile is inherited.

Model-to-text transformations. The session starts revisiting basic notions of model-to-text transformations, and then introducing MOFScript which is the language we have chosen due to its simplicity. The elements of MOFScript are illustrated using a simple, guided example. Notably, a transformation from a state machine model to the Graphviz notation to generate a visualization. Afterwards, the trainees have to generate the corresponding Java/Swing code from the intermediate models obtained with the model-to-model transformations developed in the previous session. To this end, excerpts of the expected code are given and a working model-to-model transformation is available in case some of them did not manage to make it work.

Appendix A.2. Pilot project

Once the practical part of the training is finished, it is important that the trainees put this knowledge into practice on their own. Setting up a pilot project in the company is a particularly good way of achieving this objective. For this, the research team, the trainees and some representative of the company management should meet to decide which project could be useful for them, and to scale it properly so that it is addressable within the company available resources.

In the original training, taught as part of the AutoGSA ToT project, before the pilot project stage, the developers had four weeks to implement another practical case study proposed by us, but this time they should work alone, being requested to hand in intermediate results to be reviewed. However, since this new case study had a level of difficulty significantly higher than the menu exercise solved previously as part of the training, no company was able to complete it in the planned time mainly because it was a work load too heavy. Therefore, we soon decided to cancel this part of the project, and begin the pilot project in the companies, because developers had actually acquired the competence level demanded to tackle the last phase.