Pedro Carolina

Bachelor Degree in Computer Science and Engineering

Usability aspects of Model-Driven Tools Model find and replace techniques

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Vasco Amaral, Assistant Professor,
NOVA University of Lisbon

Examination Committee

Chairperson:	Name of the committee chairperson
Raporteurs:	Name of a raporteur
	Name of another raporteur
Members:	Another member of the committee
	Yet another member of the committee

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2018

Usability aspects of Model-Driven Tools
Model find and replace techniques

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To my family.

Abstract

With the continuous increasing of software projects around the world, there is an urge to create facilities that would ease their development process. Systems design is an important phase in Software Engineering that is grounded in modelling as a way to prescribe the system at the right abstraction layer. We intend to present a solution that would help Software Engineers to deal with models implemented as diagrammatic languages, and in particular to help with the process of developing a graphical Domain-Specific Language (DSL) using Model-Driven Development (MDD) approaches. Using the current Modeling IDEs (Modelling workbenches), Models may become quite complex and large, complicating its readability for the users. The functionality to implement is expected to increase the Usability of the existing Modeling tools. This feature consists in providing to DSL users the possibility to find within a model or metamodel for specific patterns of relations among elements. There is no such implementation in any of the DSL frameworks available in the market. Textual DSL tools provide some kind of search functionality, however, these are still very primitive and do not allow to perform advanced searches. In this work, we will build a prototype of a search and replace functionality in a modeling framework to show productivity gains with respect to the state-of-the-art. We will demonstrate this with controlled experiments.

Keywords: Software Engineering, Domain Specific Modeling Language, Domain Specific Language, Model-Driven Development, Usability, Find and Replace, Controlled experiments.

Resumo

Com o contínuo crescimento de projetos de software em todo o mundo, aumentou também a necessidade de se criar instrumentos capazes de tornar o desenvolvimento destes projetos mais simples. O desenho de sistemas é uma fase importante em Engenharia de Software em que a modelação serve para definir o nível de abstração correto que o sistema deve ter. A nossa solução tem como objetivo ajudar todos os Engenheiros Informáticos que encontrem no seu ambiente de trabalho modelos implementados como linguagens diagramáticas e em particular para ajudar no processo de desenvolvimentos de Domain Specific Language (DSL) gráficas no contexto de Model-Driven Development (MDD). Ao usar-se as ferramentas atuais de Modelação, os modelos por vezes ficam bastante complexos, complicando a sua leitura por parte dos utilizadores. A funcionalidade que pretendemos implementar tem como objetivo aumentar a usabilidade das ferramentas atuais de modelação. Esta operação consistirá em proporcionar aos utilizadores das DSLs a possibilidade de procurar dentro de um modelo ou metamodelo utilizando padrões de procura, que atuarão sobre as relações entre os elementos. Não existe qualquer implementação semelhante em qualquer ferramenta de DSL disponível no mercado. As ferramentas de DSL textuais são as únicas que proporcionam ao utilizador uma espécie de funcionalidade de procura, no entanto, estas são ainda muito rudimentares e não permitem realizar procuras avançadas. Neste trabalho, vamos construir um protótipo da funcionalidade de procura e substituição numa ferramenta de modelação para demonstrar aumentos de produtividade em relação ao estado da arte. Pretendemos demonstrá-lo através de experiências controladas.

Palavras-chave: Software Engineering, Domain Specific Modeling Language, Domain Specific Language, Model-Driven Development, Usability, Find and Replace, Controlled experiments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 A Bit of History

For a long time that traditional engineering disciplines have been making use of models and modeling of various kinds (mathematical, scale, etc.) [65]. However, models in software paradigms are still generally perceived as secondary and inessential despite the fact that numerous industrial experiences have clearly demonstrated that these methodologies can improve both the quality of software and the productivity of teams that develop it (e.g.,[12, 36, 79]). There are some case studies which show positive outcomes related to the productivity gained by using MDD [5, 51]. Back in the 80's, appeared the first real attempt to shift to Model-Driven Development (MDD) methodologies as it surged the first framework in computer-aided software engineering (CASE). The goals of these tools were to enable more analysis based on graphical programs and to reduce the effort of manually coding, debugging, and porting programs. These tools aimed at reducing accidental complexity [13] in general purpose programming languages [62]. Despite the considerable effort to build an innovative tool and to start a new paradigm of programming, the CASE approach failed mainly due to the lack of usability in the tools. These also did not match with the existent underlying platforms back at that time [62] . Throughout the time some CASE problems were eventually fixed and improved, however, other issues still remain until today, such as usability, functional capabilities, scalability, etc. The complexity of these tools still make the majority of software developers to choose the path of General Purpose Languages (GPL) rather than the Model-Based [80].Despite the lack of success in the primordial times, MDD is gaining popularity. New concepts and working methodologies have been created by acknowledged technological organizations like the Object Management Group [57] (OMG).

1.2 Context & Motivation

With the increasing demand of global markets to build software applications, projects are becoming more complex [29] and therefore requiring different techniques and methodologies to bring better results. It is important that these are delivered in the proposed time and without more costs than planned. A different and promising approach is the use of abstraction methods [40]. Recent empirical studies show the adoption of Model-Driven Development (MDD) in industry [2, 37, 52, 73]. With the increased amount of projects demanded by our society, the number of errors and cost of these projects are growing. This leads to a higher pressure to improve the reliability of the software itself [65].

To achieve success in MDD and conquer more users in this type of working methodology, several things must be improved throughout the time. Below are some of the factors that are affecting the evolution of MDD:

- **Technical** - Most MDD technologies are still in early stages of development, despite existing in the industry for some years, which proves that more research is needed. There are three fundamental technical challenges that must be enhanced:
  1. **Capability** - It is urgent to solve the functional capabilities that are either inadequately developed or that do not exist at all; [65]
  2. **Scalability** - The capability to deal with large and complex system models; [65]
  3. **Usability** - Reducing the accidental complexity of developing large systems, so that they can be easily understandable by normal human intellectual capacities. [65]
- **Cultural and social** - Many developers and development managers who could benefit from MDD are simply not sufficiently informed of its advantages; The mindset is still directed on the existential pleasure of programming than the end product itself; A lack of abstraction skills, since most developers do not worry about the system architecture's as a whole system because it is not of their responsibility; [37, 65, 80]
- **Economical** - Until a team gets fully trained and capable of using MDD methodologies, it is almost certain that productivity will drop until the development teams learn how to best exploit the new technologies; Developing a software tool requires major research and development, since it must cover all the core functionalities in order to be widely used (robustness, scalability and highly usable) and also requires documentation, training materials, a maintenance team, etc. Despite all of these factors, there is still the misconception that an MDD tool should be either inexpensive or free, which leads to the construction of open source tools that are not sufficiently robust or functionally capable to cope with the challenges of the actual technological projects. [37, 65, 80]

It is agreed in the academic community (but not so much in the industrial side) that the main problems of the MDD's evolution are related to the social and organizational factors as well as the economical ones [37, 65]. Many researchers have been studying these factors and most of them aimed at the same direction: to achieve good outcomes with MDD, several attitudes inside a company must be recognized and modified [37]:

- **Adaptive**: The willingness to transform itself according to the needs and opportunities presented by adopting MDD.
- **Commited**: The company must have the willingness to provide the necessary resources to make the MDD methodology viable.
- **Iterative**: In the middle of a process there must exist awareness that a better solution may appear. The responsible people should not be "scared"to change what they have thought initially.
- **Progressive**: The self-conscious of starting small, and gradually committing more resources to adopt MDD on a wider scale.

As Software Engineers, we have to focus on contributing to the **technical factors** by providing solutions that guarantee the future of the current tools and of others that may appear.

1.3 Problem Statement

MDD has many modeling languages to express models. In this project, we will mostly be addressing to graphical Domain Specific Languages (DSL). Although there are many DSL tools available in the market, the industry is still struggling to focus their developments in this kind of technology, mainly due to a poor tool support [80]. In pursuance of a full understanding of the existing and upcoming tools, new features should be created and the available ones must be reviewed and improved.

Within sizable projects, DSLs may become very complex. This complexity can be accidental or essential when it is required for a program to function properly. Sometimes the complexity can be the essence of a program [13], hence cannot be removed. Models can have an increase in the number of elements which interact with each other in some nonlinear fashion, increasing the system's complexity much more than linearly. This can lead to an increasing difficulty to look at the relations among the entities created inside a model. The creation of an advanced *find and replace* functionality would ease the process of attenuating this complexity (accidental or essential) that models may achieve. This *find and replace* operation is weakly implemented in textual DSLs or not implemented at all in any of the graphical DSL frameworks available in the market and therefore needs to be more explored. Within a complex and large DSL, the use of textual or graphical patterns to find relations inside a model would make it easier for DSL developers to navigate

inside the defined language and explore it. This would allow them to understand how the language is built or where are the important parts of the model (entities, relations, etc.). Since a domain can get quite vast and increase significantly its complexity during its lifetime, new developers or domain experts may struggle to know the essence of the language. This can lead to a substantial amount of hours trying to understand the project rather than updating or evolving it accordingly to the company needs. This raises the complexity of communication among team members, which leads to product flaws, cost overruns, and schedule delays [13].

The current *find* and *replace* functionality available in the DSL frameworks are nonexistent or simply textual and do not allow to search in a more advanced way. For instance, it would be not possible to find a pattern of all the relations of Entity A between Entity B or C in the domain. Owing to the fact that Eclipse Epsilon is an open-source tool with a good community support, the implementation of the described functionality will be created in these frameworks. The integration of Epsilon and EMF/GMF (described in chapter 2), makes it is possible to not only implement different search patterns in a textual way but also in a graphical manner.

This solution should be able to solve one of the **capability**, **scalability** and **usability** challenges remaining in MDD world. We argue that the resolution of this technical issue can help to change positively the view that many people may still have concerning DSLs and shift their mindset towards it.

1.4 Document Structure

The current document is organized in the following way:

- Chapter 1 - **Introduction**: An overview of the dissertation is presented;
- Chapter 2 - **State of the Art & Related Work**: The current situation of the problem is presented as well as all the technologies related to it;
- Chapter 3 - **Model Search Scenarios & Overview of the Solution**: Presentation of the intended functionality and brief explanation of the solution to be taken;
- Chapter 4 - **Work Plan**: Divides the different working tasks to better optimize the available time until reaching the thesis deadline.

Chapter 2

State of the Art & Related Work

2.1 Model-Driven Development

Model-Driven Development (MDD) is a different approach to software development, characterized by using different levels of abstraction, transformations, and automatic code generation compared to traditional methods [65]. To overcome the continuous system specifications, MDD is emerging as a methodology to solve problems in a more systematic way without separating design from implementation. It is based on models as the primary artifact of the development process [28, 80].

In modeling, there are four important layers of concepts, which are crucial to understand Modeling in general. These are represented in the figure 2.1 and explained in the following list [15]:

- **Model-Based Engineering (MBE)**: MBE software models have an important role in the development process although they are not the key artifacts of it, e.g., programmers have to manually write the code based on the written/drawn models.
- **Model-Driven Engineering (MDE)**: MDE goes beyond the development activities and encompasses other model-based tasks, such as the model-based evolution of the system or the model-driven reverse engineering of a legacy system.
- **Model-Driven Development (MDD)**: MDD uses models as the key artifact of the development process. All model-driven processes are model-based but not the other way round.
- **Model-Driven Architecture (MDA)**: MDA was defined by Object Management Group (OMG) and is the particular vision of MDD, where modeling should be the key to a development process.

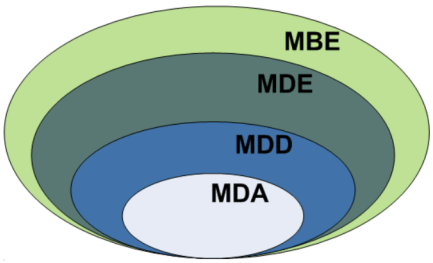

Figure 2.1: MBE vs MDE vs MDD vs MDA (taken from [4])

2.1.1 System & Model

Before explaining the definition of a metamodel, it is important to understand what a system and a model are in the modeling context.

In a more simplified definition and according to OMG standards, a model is an instance of a metamodel [76].

A model is also an abstraction of a (real or language-based) system. It is a simplified representation of a certain reality (the system), according to the rules of a certain modeling language [35], allowing predictions or inferences to be made [41].

A model is not intended to capture all the aspects of a system, but mainly to abstract out only some of these characteristics [35].

Models can be descriptive or prescriptive. A Descriptive model represents an existing system while a prescriptive model is one that can be used to (automatically) construct the target system [63] and to perform transformations. Descriptive models are also used to capture some knowledge, e.g., requirements, a domain analysis while prescriptive models (aka, "specification models"[64]) can be used as blueprints (construction plans) for system designs, implementations [41], etc. With regard to simplicity, prescriptive models will be described as models throughout this document

2.1.2 Metamodel

A metamodel can have multiple definitions according to different authors, i.e., there is not an agreement among experts. For instance, following the OMG's Meta-Object Facility (MOF) standards, a metamodel is a special kind of model that specifies the abstract syntax of a modeling language and conforms to a meta-metamodel [35].

According to [17], a metamodel is a model of a language that captures its essential properties and features, i.e., the language concepts it supports, its textual and/or graphical syntax and its semantics[35].

Others may affirm that a metamodel is a model of models [50]. Even though Eclipse Modeling Framework book [20] uses the same definition, other experts yet argue on the contrary [14].

Figure 2.2 represents the two basic metamodeling relationships. The system is represented by a model which conforms to a metamodel. This metamodel defines a language.

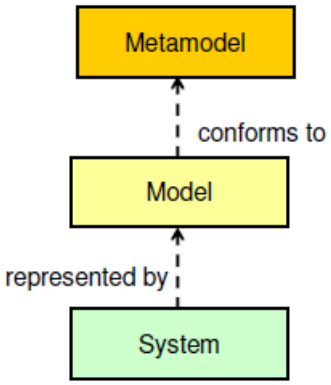

Figure 2.2: The two basic metamodeling relationships (taken from [35])

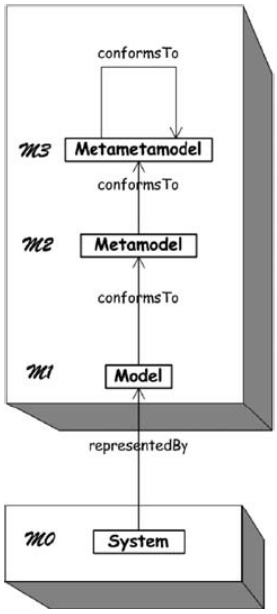

Figure 2.3: The 3 + 1 MDA organization (taken from [10])

The figure 2.3 represents the organization of the classical four-level architecture Meta-Object Facility defined by OMG. The main difference with figure 2.2 is that in this representation a metamodel conforms to a meta-metamodel, which conforms to itself. The biggest advantage of this M3 layer is the building coordination between models, based on different metamodels [10]. Model transformations and model weaving are some of the benefits of this additional layer.

2.1.3 Model Transformations

Model transformations are the heart and soul of Model-Driven Development [66]. Its purpose is to facilitate the connection between two modeling languages or to map representations of a single language. Refactoring[1] [46, 59] is one of the examples of mapping

[1] The process of restructuring and improving the quality of code (in our case, of a model)

representations within the same language. Although there are many formalisms in which a model transformation can be expressed (as represented in [19]), graph transformations based approaches are the most popular due to the level of expressiveness that they can bring [60]. In model transformations, there are two factors to keep in mind: the **source** model and the **target** model. Usually, each conforms to a different metamodel. The source model is the model that it is intended to be transformed into a different one (the target model). Using one or more models as an input and producing one or more models as an output, requires a good understanding of the semantics and Abstract Syntax (AS) of both metamodels [66] (figure 2.4). Transformations can be from model-to-model or model-to-code. In model-to-model transformations, the creation of the new model relies on the mapping of the source metamodel onto the constructs of the target metamodel [67]. Model-to-model transformations can have different transformation approaches [43]:

- **Direct Manipulation**: Accessing the API of meta-metamodel (M3) and modify the models directly;
- **Operational**: Similar to Direct Manipulation but at the model level;
- **Rule-based**: Can be graph transformation or Relational. In the latter, constraints relating source and target elements that need to be solved are defined. In the first, it is implemented the theory of graph transformation, where graphs are implemented as typed, attributed and labeled.

Model-to-code transformations generate source code that fits into an existing framework [67]. The requirement of a target metamodel, in this case, is not necessary.

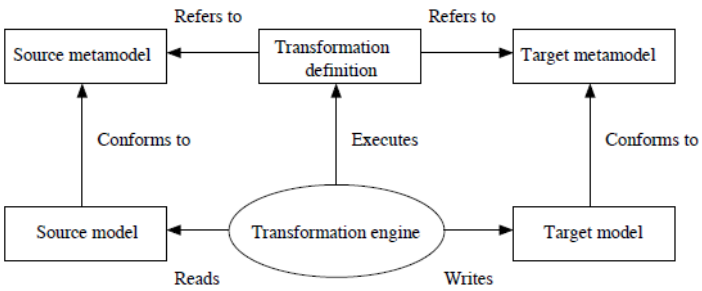

Figure 2.4: Basic concepts of model transformations (taken from [18])

In model-to-code approaches, if the model is visited in an object-oriented approach, then it was followed the **visitor-based** approach. It can also be template-based when the transformation is dependent on all of the statements of some template.

The list below presents several model transformation's use cases:

- **Model Refinement**: Used to transform a higher level specification to a lower level (although the lower specification should cover all questions from the first level);
- **Model Serialization**: Serves as a storage of some model (e.g. in a XML file);

- **Model Refactoring**: Reconstruction of the model, increasing its internal quality characteristics and without changing its behavior;
- **Model Normalization**: Has the function to decrease the syntactic complexity of models;
- **Model Parsing**: Maps concrete syntax of language back to its abstract syntax;
- **Model Migration**: When using migrations, the language defined in one model's language is changed or updated to a new language. This feature facilitates the evolution of the languages by making it possible to add new versions;
- **Model Generation**: This feature automatically generates one or more instances of a metamodel. It is quite useful because it tests the correctness of language transformations;
- **Mode Finding**: Searches for Models that satisfy given constraints;
- **Optimization**: It is another important feature which makes use of the design patterns applied in GPLs. It is useful because it may help the model design in terms of scalability, efficiency, etc;
- **Canonicalization**: It is a kind of normalization which serves to compare model equality.
- *and much more...*

With the previous list, it becomes evident that transformations are important in the context of MDD and that its benefits can be certainly valuable. These are quite useful because they make it possible to modify a model without ample changes in the implementation process (i.e., without having to manually make changes in various parts of the project, which may be very complex).

2.1.4 Model Compositions

For the sake of self-containment, although not being the focus of this thesis, model compositions will be briefly described below.

2.1.4.1 Model Merging

Model merging requires two source models to perform a transformation into a target model. Each model included in this transformation has its own metamodel. In summary, *Model A (has Language A) + Model B (has Language B) = Model C (has Language C)*. The elements present in the merged model will be exactly the same as the ones presented in the source models [43].

2.1.4.2 Model Matching

In model matching (also known as Model Weaving), the transformations create correspondence links between corresponding entities [43].

2.1.4.3 Model Synchronization

This kind of composition integrates models that have evolved in isolation [43].

2.2 Domain Specific Modeling Language

A Domain Specific Language can be characterized as a textual (DSL) or graphical (DSML) language which is optimized for a given class of problems, called a domain [63]. While General Purpose Language (GPL) users are intended to have high knowledge of technical and computational concepts (classes, events, components, variables, functions, algorithms, etc.), a DSL user's minimum requirement is that it is a domain expert, familiar with the domain concepts [6]. In this project, mostly DSMLs (graphical DSLs) will be addressed because textual DSLs already have some technologies (despite poor) related to *find and replace* functionalities. DSMLs provide transformations of models[2] which purpose is to raise the abstraction level of software and ease software development [42].

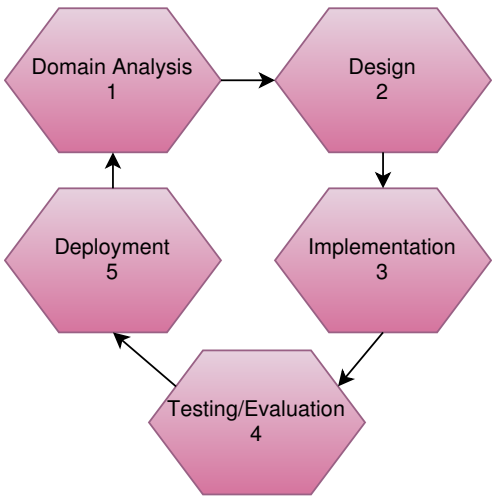

Figure 2.5: DSL development cycle (inspired in [7])

The DSML user may only have to worry about domain description, leaving complexity, design, and implementation hidden. This can improve productivity and software quality [42]. It also promotes its use by domain experts which do not need to necessarily have advanced expertise in software engineering. DSMLs present gains in productivity and reduced maintenance costs due to the expressiveness and ease of use of the domain in question. Several benefits of DSLs were represented in quantitative studies which were

[2] E.g.: model-to-model, model-to-text, text-to-text, text-to-model

conducted by [8, 34, 39, 44]. To develop a good DSML, it is advisable to process all stages represented in figure 2.5.

2.2.1 Domain Analysis

This phase is the most important of all the DSL life cycle development because it is where is analyzed the domain and gathered all the important knowledge related to it. The study of the domain by experts usually leads to a more rigorous design. Any wrong decisions made in this initial phase will make it harder to change further details in the development process of the DSL life cycle. The result of this phase will be a domain model [78], which defines the concepts, properties, and vocabulary within the system. In the end of this phase, artifacts such as design documents, requirement documents, user manuals, etc, should be completed for the next stage of the development process.

2.2.2 Design

To define the language design, it is necessary to use a metamodel to build the syntax. This **syntax** can be **concrete** (CS) or **abstract** (AS). An AS characterizes elements of a domain, their relationships, and properties. The CS can be characterized as the representation of a DSL in a human-usable form [42]. The CS represents the interaction that the user is able to have by making use of the rules expressed in the AS. CS has a type of representation(text, graphic, etc.) and a style (i.e., declarative or imperative). The semantics is another important concept that must prevail in the design of the language. It defines the behavior and the meaning of any of the constructions defined in the language.

2.2.3 Implementation

In this phase, all the relations defined in the metamodel are translated into native code. This process is automated using tools such as MetaEdit+ [48], GMF [30], Obeo Designer [56], Microsoft DSL Tools [49], EuGENia [27], etc. Unless the generated code has to be changed manually by a programmer due to lack of expressiveness, the process of automation saves time and provides consistency to the generated software. Textual or graphical editors are also generated in this phase, easing the process of development for the DSL user.

2.2.4 Testing/Evaluation

2.2.4.1 Testing

The main objective of this phase is to verify the conformity of the requirement models described in the domain analysis. The use of experimental processes, heuristics and questionnaires may help to identify the means to evaluate the quality in use of the created language. Testing is used to verify the syntax, system restrictions and the semantics

of the developed language [63]. Syntax errors can be found whenever there is lack of elements to express a specific language. In other words, for a language to be complete and able to completely satisfy a specific domain, all elements which belong to that domain must conform and exist. To test the system restrictions, a user can give an incorrect input on purpose to some element and perform a quick validation. Semantic errors are encountered each time the program has an unexpected behavior from which the one that was expected from the DSL creators.

2.2.4.2 Evaluation

The evaluation part has an important weight for the matters of this research project. DSLs intentions are to favor an efficiency's increase of its users without having to cause extra organizational costs, inconveniences, dangers, and user dissatisfaction, undesirable impacts on the context of use, long periods of learning, assistance, and maintenance [16]. All of the stated above point in the same terminology: **Usability**, which goal is to achieve the **Quality in Use**[38]. The ISO IEC 9241-11 (2001) standard defines Usability as the "*extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use*"[7]. Although the importance of evaluation, many people still consider it a waste of time and resources and they prefer to risk using or selling inadequate DSLs [6]. There are some aspects to evaluate Quality in Use of the DSLs [6]:

- **Effectiveness**: Determines the preciseness to complete a language sentence;
- **Efficiency**: Tells the level of effectiveness achieved according to factors like time, cost, mental and physical effort;
- **Satisfaction**: Provides the happiness of the user towards the use of the implemented language;
- **Accessibility**: Focus on learnability and memorability of the language terms.

There are also different steps that must be taken in order to completely evaluate a language in the correct terms, as observed in figure 2.6. In the **Subject Recruitment**, users must be organized in groups according to the intended objective of the evaluation. After, in **Task Preparation**, occurs a selection of which actions need to be made to provide the proper results. The **Pilot Session** serves as a "mockup Exam", allowing to test if all the necessary materials were gathered. The **Evaluation Session** is where the users get their first contact with the DSL and try to understand its concepts before taking an Exam. The results from the exam are collected for further analysis in order to understand how easy it was for the users to grasp what was made. Despite having the results from the exam to make some inferences, the users are also invited to respond to a **Final Questionnaire**. In this, users can send a feedback about the DSL, concerning qualitative aspects. The end

of the evaluation process always have a **Analysis of results** of all the aspects previously studied.

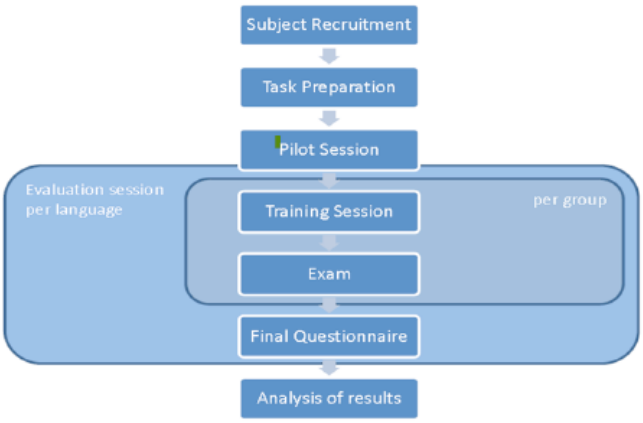

Figure 2.6: DSL evaluation's process steps (taken from [6])

2.2.5 Deployment

Represents the last phase of the DSL life cycle (unless the domain suffers modification, thus reinitiating with *Domain Analysis*). The process of deploying is similar to the deployment of any other software.

2.3 Language Metamodeling Workbenches

Language Metamodeling Workbenches (LMW) are tools that provide high-level mechanisms for the implementation of domain-specific languages by providing the required functionalities to define a metamodel that establishes relations and rules in the models to be created. LMWs enable a development of new languages to be affordable by supporting efficient definition, reuse and composition of the languages and their IDEs [26]. As we will see in the subsections below, LMWs exist in many different flavors even though all of them have the same goal in common: to facilitate the development of DSLs. The LMW to work with will be better if it covers the feature model represented in figure A.1, proposed in the annually discussed in the Language Workbench Challenge (LWC) [26].

2.3.1 Eclipse Modeling Framework & Graphic Modeling Framework

Eclipse Modeling Framework [23] (EMF) as the name suggests is a framework within the Eclipse platform and is a code generator facility that lets the user define a model. EMF metamodel is named **Ecore**. EMF model glues Java Interfaces, XML or UML together, as shown in the figure 2.7, making it possible to interchange to any of these previous forms. Graphic Modeling Framework [30] (GMF) is also a framework within Eclipse platform which purpose is to generate and manage all graphical editors **based on the**

EMF metamodel. Both EMF and GMF are available for Windows, Linux, and Mac OS. There is a considerable support concerning these frameworks documentation in papers, tutorials, books, guides, presentation/ workshops, contributed articles, etc. The graphical component is highly customizable, being possible to modify elements and add or remove new ones. It also provides model composition facilities.

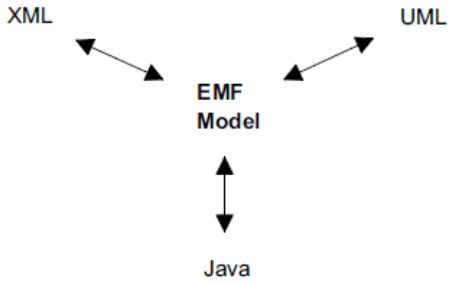

Figure 2.7: EMF unification (taken from [20])

2.3.2 Epsilon

Epsilon [25] is a powerful framework available for Windows, Mac OS and Linux and all the questions that may appear on this technology are richly documented. It is built under Eclipse platform and integrates with EMF/GMF, serving as a family of languages and tools, represented below:

- **Epsilon Object Language (EOL)**: EOL is an imperative programming language which combines both JavaScript and Object Constraint Language [61] (OCL) for creating, querying and modifying EMF models.
- **Epsilon Transformation Language (ETL)**: It is a language built on top of EOL which provides model-to-model transformations.
- **Epsilon Validation Language (EVL)**: EVL is also built on top of EOL. It is a validation language integrated with EMF/GMF which constraints are quite similar to OCL, yet more powerful since dependencies between constraints and customizable error messages are available to users.
- **Epsilon Generation Language (EGL)**: EGL is a template-based model-to-text language for generating code, documentation and other textual artifacts from models.
- **Epsilon Comparison Language (ECL)**: ECL is a hybrid, rule-based language for comparing models.
- **Epsilon Merging Language (EML)**: EML is a hybrid, rule-based for merging homogeneous or heterogeneous models.
- **Epsilon Wizard Language (EWL)**: EWL is a language tailored to interactive in-place model transformations on user-selected model elements.

- **Epsilon Flock (EF)**: EF is a model migration language built atop EOL, for updating models in response to metamodel changes.
- **EuGENia**: Tool which helps developing DSLs by automatically generating GMF models required to the implementation of the GMF editor, using Ecore's EMF.

The platform to implement the *find and replace* functionality will be Eclipse Epsilon.

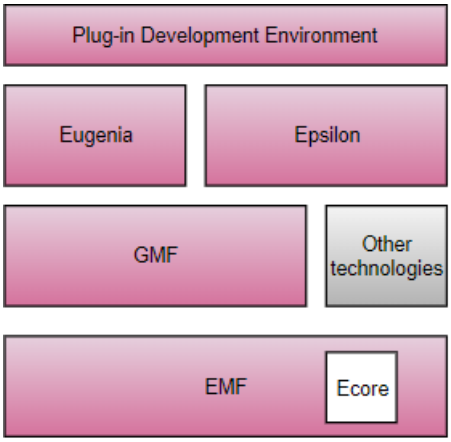

Figure 2.8: Architectural view of Epsilon integration with the required technologies

In the figure 2.8 is represented the architectural view of the frameworks that will play a major role in our solution. Epsilon, with the support of EuGENia, will automatically generate the GMF editor which conforms to EMF's Ecore metamodel. Plug-in Development Environment (described in section 2.4) will serve to create the plug-in that will retain the core of our solution (described in chapter 3).

2.3.3 Generic Modeling Environment

Generic Modeling Environment [77] (GME) is a configurable toolkit accomplished by metamodels to specify the application domain, only available for Windows platform. Even though documentation is not so rich as compared to other languages, there are still some tutorials and documents that help to learn about this tool. The metamodeling language is based on the UML class diagram notation and OCL constraints. The metamodels generate the code of the target domain-specific environment that gets stored either in a model database or in an XML format. After, these models are used to automatically generate the applications. The transformation languages are possible by using Graph Rewriting And Transformation (GReAT).

2.3.4 MetaEdit+

MetaEdit+ [48] is a Domain-Specific Modeling solution that supports multi-user and multi-platform environments. The documentation is broad, existing a lot of support for beginners and more advanced users. It is possible to utilize transformations on models.

The language concepts, their connections, and associated rules are illustrated in a notation. By drawing the symbols, the full graphical behavior is provided automatically. The graphics customization is reasonable.

2.3.5 Obeo Designer

Obeo Designer is powered by Eclipse Sirius [56] and is available for Windows, Linux, and Mac OS. Users can use a set of editors (diagrams, tables, and trees) to create and edit models in a shared repository. This tool is integrated with GMF to manage the graphical part and with EMF to work with the models. This LMW is well documented and also allows transformations on models. It is open source, which is a plus.

2.3.6 AToMPM

AToMPM [54] is primarily a graphical modeling tool. On the concrete syntax side, all model elements are displayed as SVG [71]. All model transformations are based on T-Core [72]. It is independent of any operating system because it runs solely over the web. Multiple users can be logged in simultaneously to work and share modeling artifacts [71].

2.3.7 XText

Xtext [83] is an open-source tool that can be used in the Eclipse IDE. The documentation available is sufficient to learn easily how to use this language as there are also training available to the more interested users. It is compatible with graphical editors like Sirius [56] or Graphiti [33], allowing to perform graphical customization. It allows model transformations, constraint checking, and code generation [70].

2.3.8 Textual Editing Framework

Textual Editing Framework [9] (TEF) is an Eclipse plug-in that allows the user to create textual editors for EMF-based languages. It has a syntax definition language called TSL (Textual Syntax Language) which describes a textual notation for a given Ecore meta-model. From TSL it is possible to automatically generate a TEF editor. TEF Also creates facilities to DSL-to-model transformations, by creating a parser which is interpreted at runtime [47].

2.3.9 EMFText

EMFText [24] is an Eclipse plug-in that allows developers to define textual DSLs fast and without the needs to learn new technologies and concepts. EMFText allows specifying a concrete syntax for an existing EMF model [47]. The generated editor supports outline view, customizable syntax highlighting, code completion, bracket handling. A lot of documentation is available on the internet. Due to the fact that EMFText implements

EMF resource API, model transformations are possible when using Modeling Workflow Engine (MWE).

2.3.10 Language Metamodeling Workbenches Comparison

The table 2.1 and the previous subsections were inspired in [3].

The criteria used for these comparisons were:

- **None**: When the respective component is nonexistent;
- **Good**: When the component exists but is not so rich as it could be when compared to the other evaluated tools;
- **Excellent**: When the component exists and it provides a robust structure.

Evaluation Criteria	EMF/GMF	Epsilon	GME	MetaEdit	Obeo	AToMPM
Platform	●	●	◑	●	●	●
Documentation	●	●	◑	●	●	◑
Multi-user	○	○	○	●	●	●
Graphics Customization	●	●	◑	◑	●	◑
Transformation	◑	◑	◑	◑	●	◑

Table 2.1: Comparison among Graphical DSL tools
(Subtitle: ○ - None; ◑ - Good ● - Excellent)

Evaluation Criteria	XText	TEF	EMFText
Platform	●	●	●
Documentation	●	◑	●
Multi-user	○	○	○
Editor Customization	●	◑	●
Transformation	●	◑	◑

Table 2.2: Comparison among Textual DSL tools
(Subtitle: ○ - None; ◑ - Good ● - Excellent)

Before analyzing in detail the tables above (2.1, 2.2), it is important to mention that obviously that there are more DSL tools available in the market. Nonetheless, the chosen ones serve well the purpose of analyzing the differences that these tools have among each other. These are evolving and being improved throughout the time. Certainly, some fulfill the specifications of some projects better than another, depending on the context. Each of these frameworks allows creating models based on a language. Starting by analyzing the first table (2.1), we can observe several differences that these tools provide when building a DSML. *GME* is the only tool of the mentioned above that supports only one Operating System (Windows), hence more limited. *MetaEdit+* is a commercial tool which offers a good multi-user platform, as well as *Obeo Designer* and *AToMPM*. *Obeo* is the only

tool with a specific language for transformation (**Acceleo** [1], based on the OMGL MTL standards [58]), whether other tools transformation's have to be manual (e.g. *Epsilon*), via SOAP/Webservices API (e.g. *MetaEdit+*) or by other means. The tools that use *EMF* as a common technological base (*GMF*, *Epsilon*, *Obeo*) can be considered as the most complete and functional for the purpose of generating DSL graphical editors, as shown in a systematic review [32]. *AToMPM*, despite not being the richest tool available (lack of documentation compared to others, composition, etc.), is a motivational reason to our intentions on building a new plugin. This tool was born as a research project from many academic researchers [53] and throughout the time its improvements are noticeable. *EMF/GMF* becomes a more powerful tool because of the integration with *Epsilon* and vice-versa. For instance, despite *EMF/GMF* not supporting composition, the integration with *Epsilon* turns it a possible feature. *Eclipse Epsilon* is the tool in which the intended functionality of this thesis will be implemented, on top of *EMF and GMF*. *Epsilon* has a vast community support, technologies integrated within and it is an open source tool. All of the previous factors and continuous substantial usage of this framework made it the chosen DSL tool to work with. The table 2.2 shows a brief comparison of three well known textual DSL tools. Textual DSLs retain several advantages such as merge and version control, code completion, error markers, quick fixes, easy information exchange (e.g., via email) [47], etc. Both *XText* and *EMFText* offer similar editors customization which can be extensible by developers, opposite to TEF [47]. In terms of model transformations, *XText* provides an integrated model-to-text and model-to-model frameworks (*Xpand* [81] and *Xtent* [82] respectively). Model transformations in *EMFText* and *TEF* must be configured with other frameworks. In resume, both textual or graphical DSL tools are constantly evolving and gaining new features throughout the time. The use of each depends mainly on the user's preference. In my opinion, graphical DSLs have more potential because they give the possibility of a domain expert (with or without informatic skills) to easily construct its language and model instances of that same language.

2.3.11 *Find* and *Replace* State in Language Metamodeling Workbenches

As already proven, the *search* operation is among the most powerful innovations of our time [55]. It is used in many different categories such as web, e-commerce, enterprise, desktop, mobile, social, and real-time search, and discovery. In a DSL framework context, it would mean the capability that a user would have to find answers within a model, i.e., a specific element or diagram. It could also help a new user to explore a model in order to understand the relations between elements. When leading with small or medium models, manual search or plain memorizing will certainly be more effective than using a query. The necessity of this operation only becomes crucial when facing a complex and large model. In this situation, a user would spend a lot of time to *find* the desired elements. Obviously that a programmer could find a turnaround to this situation and create a program which would define specific search patterns to *find* the pursued elements.

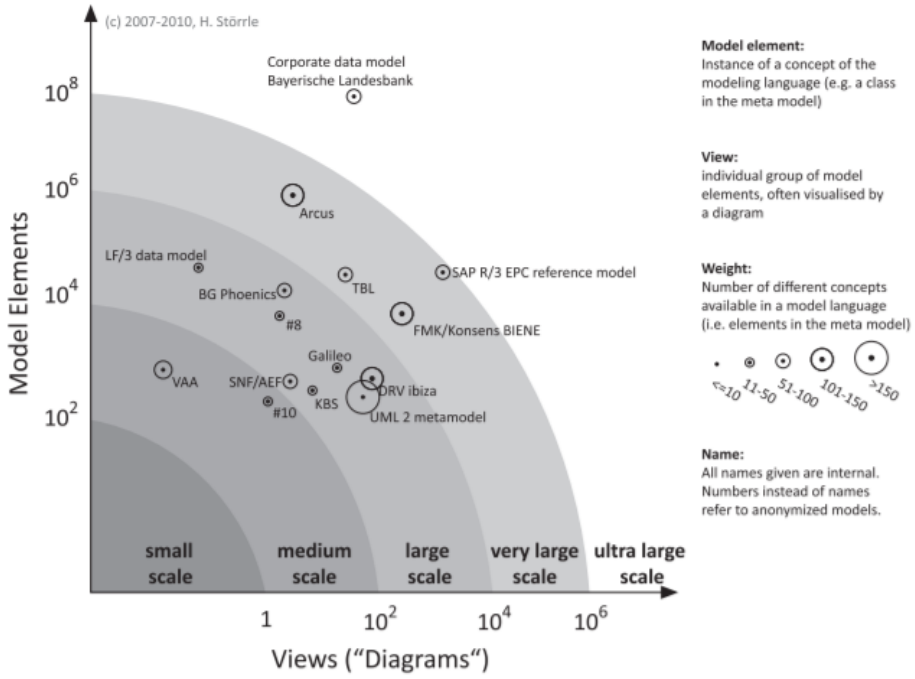

Figure 2.9: A survey of large-scale models. The size of the circles ("Weight") shows the number of concepts in the respective modeling languages (Taken from [68])

This is not a good solution because DSL users do not necessarily need to be only programmers (they can be Domain experts, etc). The figure 2.9 perfectly states how enormous a model can become, hence hard to explore and *find* the necessary model elements. A solution to this problem was addressed in the Visual Model Query Language [69] (VMQL) project. In this research project, it is possible to use queries to find model elements, in this case, UML elements. The figure 2.10 shows an example of the VMQL project in action. In this, the user searched for a Product Class. The search engine found this class defined in the model and navigated to the respective part, highlighting the expected result.

Unfortunately, for DSL tools, there is not any similar solution, **yet**. The table 2.3 addresses this lack or poorly implementation of the feature *find and replace* in some of the available DSL frameworks. Only the textual DSLs have some kind of search mechanism implemented for model searching. However, these are very rudimentary and do not allow to perform an advanced search pattern to look for relations inside the model. They merely let to look for a full-text input, i.e., an exact word must be given. An advanced *replace* operation is also an important feature that we want to add to these tools as it can increase in quality the refactoring process of the selected models. Both *find* and *find and replace* operations might be based in model transformations. By having a deeper understanding of these, a user can tell that model transformations procedure is to find some model elements and update them. The process of our *find and replace* operation should not differ a lot from the model transformations algorithm, as each of these iterates through the model elements.

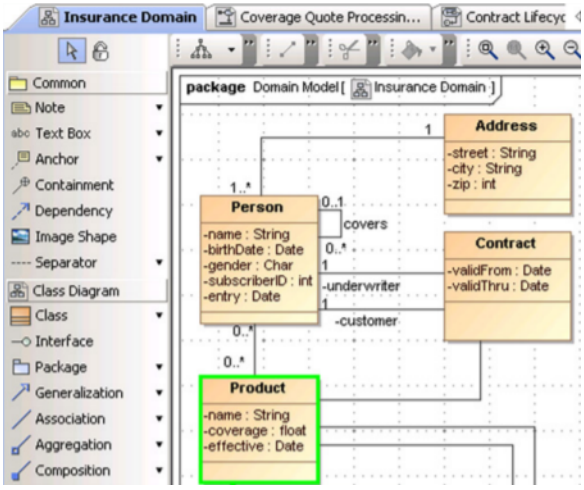

Figure 2.10: UML Class Diagram - Use of *find* operation in VMQL project (Taken from [68])

We strongly believe that the implementation of *find* and *replace* could ease the understanding of complex DSLs in terms of **capability**, **scalability** and **usability**, as mentioned in the introductory chapter (1). We believe that each framework would benefit from this feature because it enhances not only the readability and construction of the DSL but also the time of maintenance of the specific language.

Evaluated Platform	Textual	Graphical
EMF/GMF	○	○
Epsilon	○	○
GME	○	○
MetaEdit+	○	○
Obeo	○	○
AToMPM	○	○
XText	◑	○
EMFText	◑	○
TEF	◑	○

Table 2.3: Comparison of *Find and Replace* functionality among DSL tools (Subtitle: ○ - None; ◑ - Poor ● - Excellent)

2.4 Plug-in Development Environment (PDE)

Plug-in Development Environment [21] (PDE) provides tools to create, develop, test, debug, build and deploy **Eclipse** plug-ins. As our solution to the problem will be created within a plug-in, it is also important to mention how these are built and integrated with each other. It is crucial to know how plug-ins work and how they can be compatible with other plug-ins. The use of plug-ins contributes to loose coupling through the mechanism of extensions and extensions points. There are four important notions to understand in

the plug-ins development context, described below [45]:

- **Dependencies**: A plug-in may have a list of Dependencies, i.e., all the plug-ins that in some way can contribute to some other plug-ins. These dependencies can also be necessary for some plug-in to compile, as it can require some of its functionalities.
- **Extensions**: An Extension is a contribution that a plug-in can make to another. This can be complemented by adding it to an Extension Point.
- **Extension Points**: The creation of an Extension Point requires the creation of a schema (PDE provides an editor for the composition of the latter). This schema must be followed by all the clients in order to be correctly processed during runtime.
- **Runtime**: The Runtime serves as the process to set the library's type, visibility, and content of the plug-in.

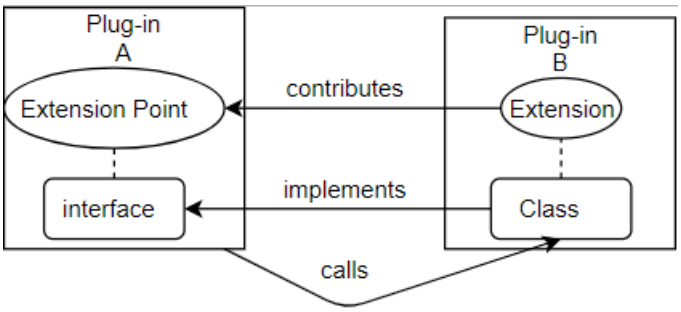

Figure 2.11: Plugin architecture (inspired in [11])

Figure 2.11 summons up in a simplified manner the most important relations that two plug-ins may have. Let us suppose that it is intended to implement a new Plug-in B to integrate with one another A. The class of Plug-in B must implement the interface displayed in A to avoid compatibility issues. We can think of the Extension Point to be a socket and the Extension to be a plug. There is a wide variety of sockets available out there, but only the adequate extension will be compatible with it. The same happens in the plug-in environment. For two plug-ins to be compatible, the Extension of the plug-in intended to be integrated with another must be compatible with its Extension Point.

Concerning our research project, there are two very important plug-ins which will have to be used within our solution:

- **org.eclipse.emf.ecore**: Can be seen as the meta-metamodel of all the metamodels and models to be defined. It contains all the so-called Ecore Components which implement *EObject*, the main element of the Ecore Modeling Framework. All of the elements and hierarchy relationships among these can be seen in the figure 2.12. This plug-in also contains Java Language Types such as EBoolean, EDouble, EFloat, EInt, EString, etc. It also has external types like EDate, EEList, EFeatureMapEntry, etc.

- **org.eclipse.core.runtime**: This plug-in provides support for the runtime platform, core utility methods and the extension registry.

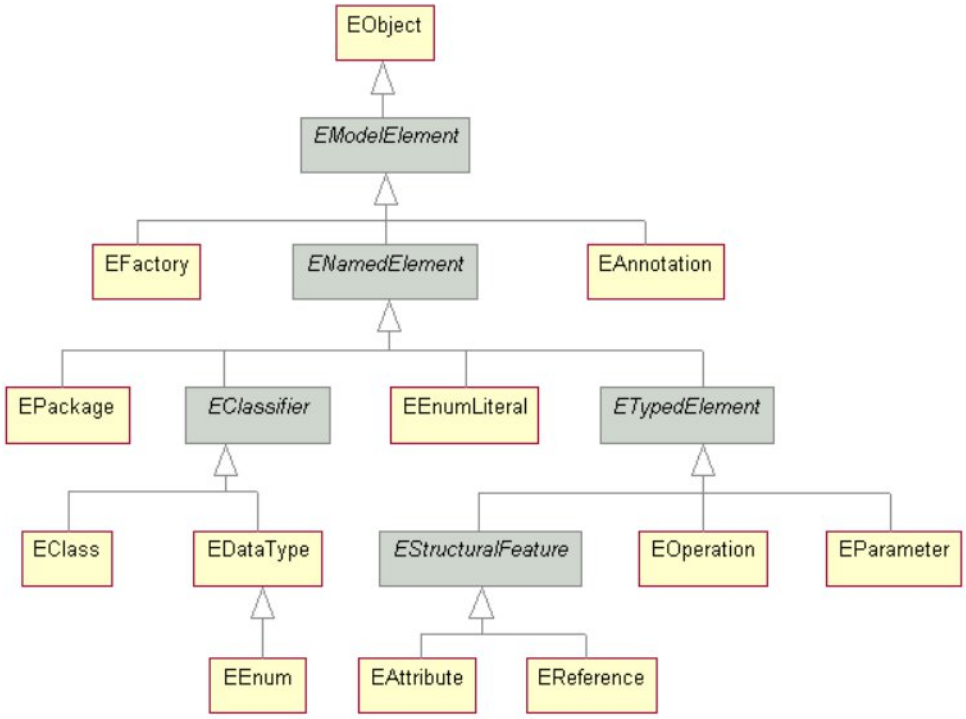

Figure 2.12: Overview of *org.eclipse.emf.ecore* elements (taken from [22])

2.5 Survey

A survey holding **25** respondents was conducted with the goal of understanding the general knowledge that software developers have concerning MDD. It also served as a study to know if developers think that a *find and replace* functionality in a DSL could bring benefits in its life cycle process. An univariate analysis and bi-variate analysis were used to study the answers from the respondents. The conclusion taken from this research study is that the people which are familiar with the MDD technology think that it is useful to create a *find and replace* functionality which could bring many advantages in the creation and maintenance of DSL projects. A general approach still must be taken (by companies, universities, etc.), because there is a good percentage of people that do not remain aware of the power that MDD can achieve in the use of DSLs. The complete study (*Introduction*, *Background*, *Objectives*, *Survey Method*, *Survey Results*, *Threats to Validity and Conclusion*) can be seen in **Annex** I.

Chapter 3

Model Search Scenarios & Overview of the Solution

This chapter proposes graphical *find* and *replace* patterns for DSMLs. These will be presented in several mockups to exemplify what the user may expect from this up-and-coming feature. An overview of the solution will also be discussed.

3.1 Scenarios

Similar to VMQL [69] project, the plan is to provide to the user a kind of editor to perform the *find and replace* operations, working together with the modeling editor (as well as having almost the same design). In this way, the user familiarity with the new technology would be faster introduced to the user. Despite the two editors being visually separated, they would be intrinsically connected. Each time a user introduces a new *find* operation or tries to *replace* some elements in the newly created editor, the modeling editor would show the results in a highlighted manner (examples shown in section 3.2.1 and 3.2.2).

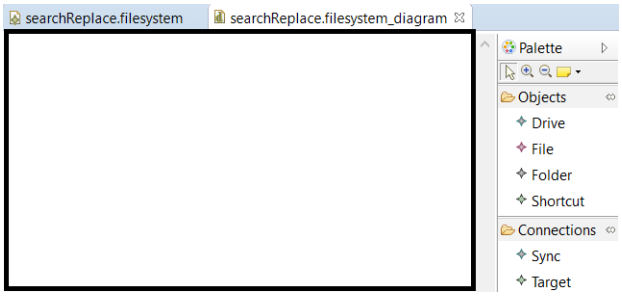

Figure 3.1: Editor environment's mockup for the *find and replace* functionality

Figure 3.1 represents a brief look of how a *find and replace* editor design would seem like. It is quite straightforward its similarities with the modeling editor. On the right side

it is possible to notice the language defined by the metamodel. The user can drag any of the represented elements in the language and drop them inside the black window. The results of these searches should be able to match direct matches or similar matches in the modeling editor, by highlighting the results. In case a search was unsuccessfully achieved, the user would be accordingly notified about it. For further research, upon a failure *find*, a tip could be given to the user on how to possibly create a better pattern to look for the desired elements within the model. These type of *graphical find* provides an easier model navigation and exploration to the DSL user. It is suitable not only for programmers but for all the people which are familiar with the model domain concepts. Thereby, more people would be qualified to be inserted in DSL projects and help to build or maintain one or more models. The *find and* **replace** operation would also serve as refactoring model purposes or to solve errors committed by the DSL users. In this replacement process, to *find* the desired element would be exactly the same as we are planning to. To *replace* an element, the user would have to pick a matching element from the modeling's editor and try to *replace* it with the element provided in the *search editor*.

3.1.1 Considerations

There are several situations in which the *search* patterns might lead to unintended results. A different approach to the problem could have been answered by using Object Constraint Language (OCL). It is a declarative language used to impose restrictions on paths of associations, universal properties, and existence states properties [61]. This language provides a wide set of queries to intervene with models. So, why are we not considering the use of OCL? As described along this document, our intentions are not only to provide an important functionality to the DSML context but also to achieve that goal by keeping in mind Quality in Use concerns (described in chapter2). OCL would complicate too much the creation of *search* based queries, especially when used by DSML users that are not so familiar with programming concepts. Also having in mind usability concepts, the best possible number of scenarios must be analyzed. For instance, what if a search pattern shows 10 different results spread along the model? How would the user be able to study these without having to scroll all the way around to find them? One possible solution would be to provide a list of results, that on-click would redirect the user to the highlighted result. Most certainly that more situations will appear and require its particular attention.

3.2 Mockup's representative Model & Metamodel

The use of prototyping simulates important aspects of the main system, while not being necessarily bound to it by the same hardware speed, size or cost constraints [13]. Mockups are a good way to prototype the possible solution because they can show an approximate design on how the system will be represented after the intended operation

has been implemented (changes may happen, a mockup can be seen as a draft). It also serves as a good way to receive feedback from the users before any further procedure is taken, avoiding unnecessary changes in late stages of the process or at the end of the development.

The following metamodel [75] will be used to build a model to represent our mockups:

```
@gmf.diagram
class Filesystem {
    val Drive[*] drives;
    val Sync[*] syncs;
    }
class Drive extends Folder {}
class Folder extends File {
    @gmf.compartment
    val File[*] contents;
    }
class Shortcut extends File {
    @gmf.link(target.decoration="arrow", style="dash")
    ref File target;
    }
@gmf.link(source="source", target="target", style="dot", width="2")
class Sync {
    ref File source;
    ref File target;
    }
@gmf.node(label = "name")
class File {
    attr String name;
    }
```

The metamodel above represents an elementary vision of a computer file system. For simplicity's sake, let us consider that there can be no element with the same *name* identifier. With the given objects and connections, a lot of different file systems architectures could have been created.

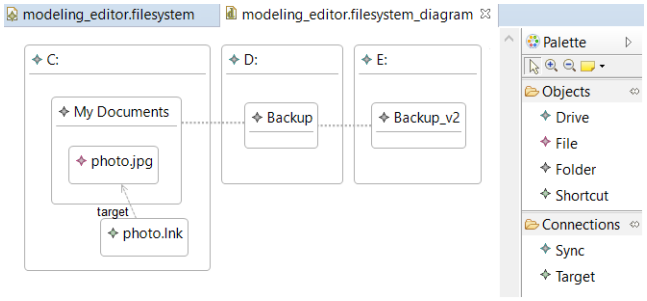

Figure 3.2: Filesystem's modeling editor (adapted from [25])

The created model (figure 3.2) contains three drives that contain folders which contain files. *Drive D:* has a special folder that syncs with another folder of *Drive C:* to create

backups. The same happens with *Drive D: and E:*. A shortcut is also represented in *Drive C:*.

The following mockups will always consider the model represented in figure 3.2.

3.2.1 Graphical *Find* Scenarios

In the graphical *search* editor, a drag-and-drop functionality is to be implemented, likewise the modeling editor. The DSL users will have to drag a specific element and create different patterns with it, e.g., by adding relations to that element. In the following *search* scenarios, we will demonstrate some of the possible *search* patterns that this functionality can bring to help in the exploration and navigation inside large models. Opposite to a textual *search* scenario, in the graphical *find and replace* implementation, users do not need to know any extra notation types rather than the ones used in the modeling editor. This brings benefits to the tool in terms of **usability**. These patterns can include **classes**, **attributes** or **associations/references**.

In the first example (3.3), the DSL user dropped the Metamodel's *Drive class* within the *search* editor and used the exact name to *find* that element, i.e., by using an *attribute*.

Observing the figure 3.4, indeed we can state that an element with the *attribute* "D"existed in the modeling editor. Therefore, the user is redirected and a highlighted result appears in the respective element to represent a successful search.

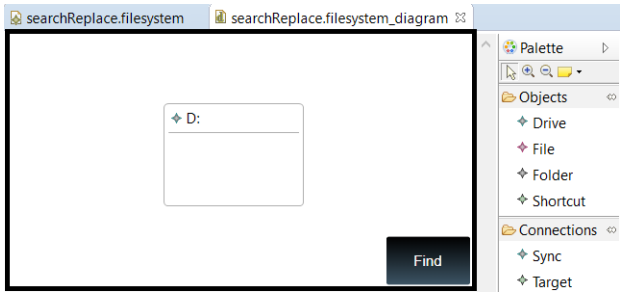

Figure 3.3: *Find* a Metamodel's element using a *class and attribute*

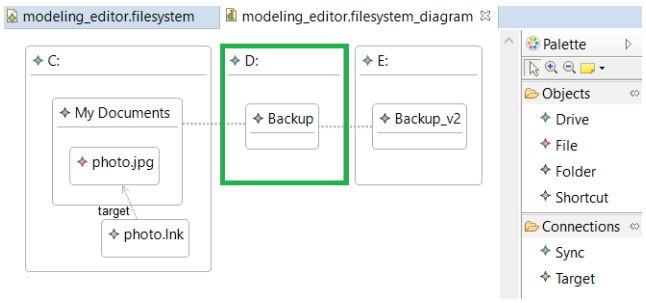

Figure 3.4: Highlight of a specific element

In the second example (figure 3.5), the DSL user did not make any changes after dropping the element within the *search* editor. This means that the user intentions were to

use a general Metamodel's *Drive class* without the use of any attributes or references. This kind of operation will look for **all** the classes represented in the model which correspond to that Metamodel's class. The figure 3.6 shows the result upon inserting the class *Drive* within the *search* editor (figure 3.5). The highlighted sections contain all the *Drive classes* defined in the model.

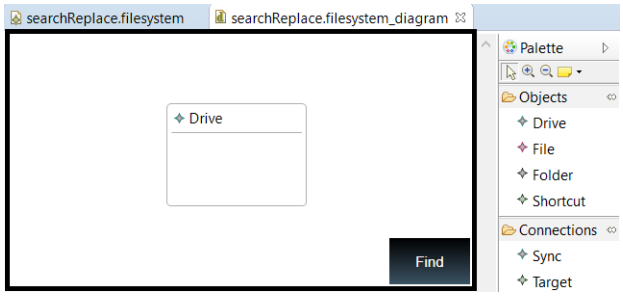

Figure 3.5: *Find* of any Metamodel's elements corresponding to a specific class

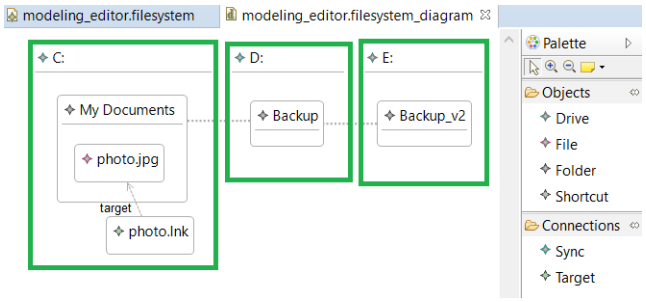

Figure 3.6: Highlight of all the class elements

In the third example (figure 3.7), the DSL user utilizes the combination of two Metamodel's Classes (*Drive* and *Folder*) together with wildcard textual *search* on the *attribute*. In this situation, the DSL user is looking for inheritance results as *Drive* extends *Folder*.

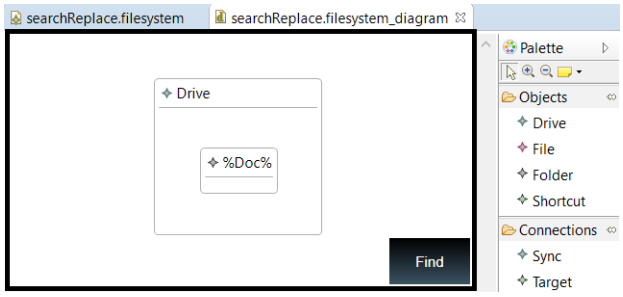

Figure 3.7: *Find* of an element contained in another (inheritance)

The result of the previous *find* operation should take the DSL user to a similar situation as the one represented in figure 3.8. It is important to note that this pattern's results could *find* more than one element, as the use of the wildcards could encounter more element *Folders* containing the the same attribute name "Doc"simultaneously.

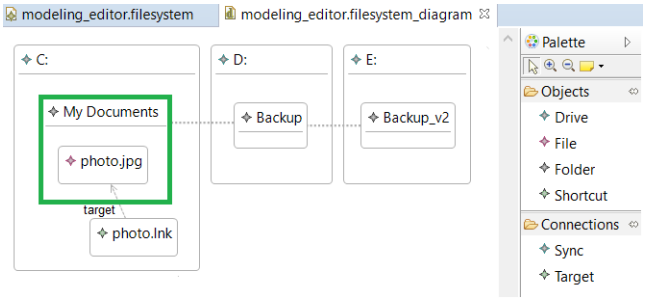

Figure 3.8: Highlight of an inheritance result

In the figure 3.9, the user expects to find a relation among two different Metamodel's elements *Drive* classes, using their name identifier to perform the *find association*. Looking at figure (3.10), it is perceived that an **indirect** association between elements is achievable.

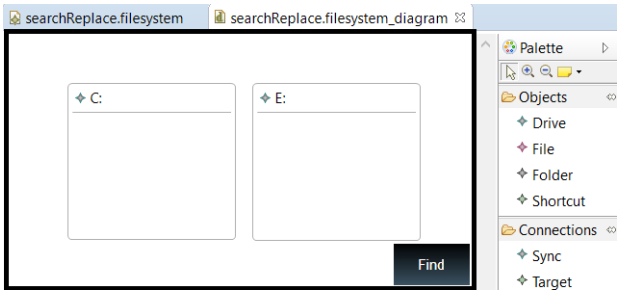

Figure 3.9: *Find* of relation among two different elements

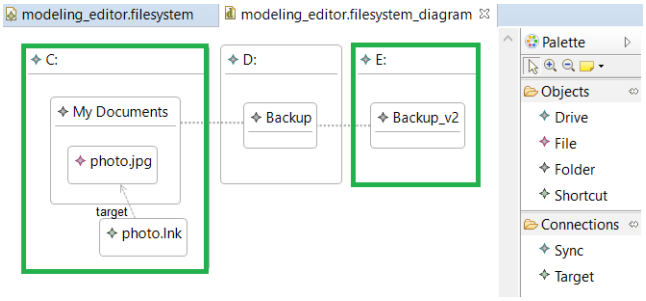

Figure 3.10: Highlight of a successful indirect association

3.2.2 Graphical *Replace* Scenarios

The *Replace* operations will also take an important role in our final solution. When the intentions are to simply *replace* an element directly for another, it is trivial since we only have to check if the existent connections match the new element. The complexity raises when the DSL user wants to use a **replacement pattern**. What happens to the old connections of the replaced element? What happens when the *replaced* element has new *references* to other elements? How can the DSL user specify which references he plans to

maintain or to eliminate? How to overcome all of these maintaining the **Usability** of the tool? These are some of the meaningful questions that must be analyzed carefully. It is important to remind that our goal, if possible, is to prevent that the **essential** part of the solution becomes too complex to the DSL user.

The first part of the process is based on using the *find* operation described previously, by looking for the desired elements. By submitting the pattern, the user would navigate to the modeling editor's highlighted element(s) (figure 3.11). From there, the user would select (e.g., by pressing *CTRL+SELECT*) the elements to take part in the *replacement* pattern, highlighting these. When decided, the user would be redirected to the *search editor* containing all the selected elements (as well as the references that these may have - figure 3.12). Only the selected elements can be modified, **preventing** undesired results to those who kept unselected (e.g., *Drive* "C:").

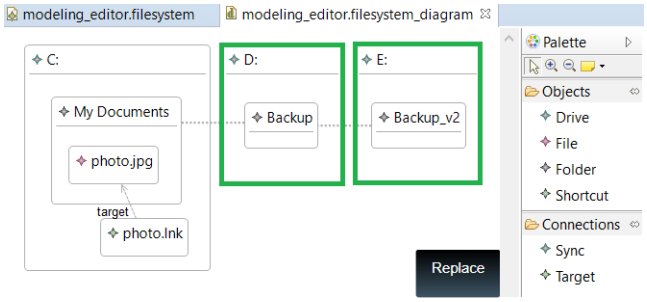

Figure 3.11: Replace - Initial Step

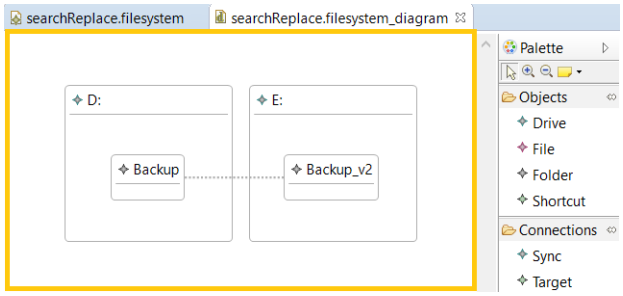

Figure 3.12: Replace - Intermediate Step

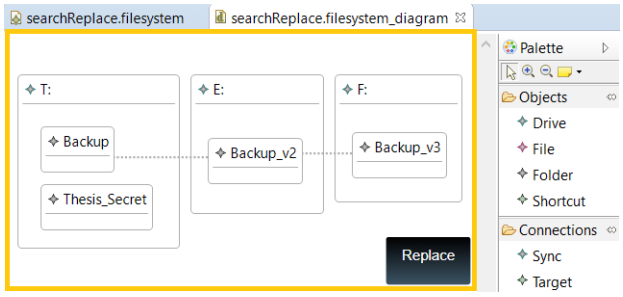

Figure 3.13: Replace - Final Step

All of these transformations (figure 3.13) would be reflected in the modeling editor along with the unmodified *Drive* "C:", which would keep connected to *Drive* "T:", before named "D:".

3.3 Overview of the solution

The implementation of our solution will rely on the creation of an Eclipse plug-in utilizing Plug-in Development Environment (PDE) and interconnecting it with other *EMF* plug-ins. The plug-in will be based on one of the next options:

- **Scratch Search Algorithm**: As the name suggests, an algorithm from scratch would have to be created. Possibly using or adapting one of graph's theory algorithm to iterate through the DSL editor. This iteration would be based on the pattern submitted by the user in the *search* editor.
- **Model's Transformation Algorithms**: On existing model's transformation, all the elements, properties and relations are processed through an algorithm before being converted into the target model. Thereby, in some way, a sort of *find* has to occur to look for all the elements within a DSL editor's model. A partial implementation of these model transformation's algorithm could be adapted to our solution from the existing implementations (e.g., provided by ETL engine).

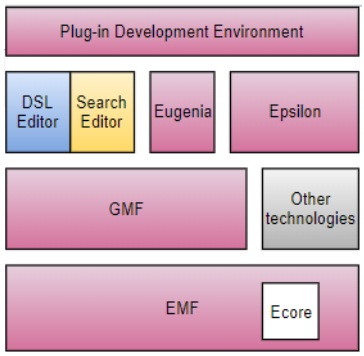

Figure 3.14: Architectural view of the solution

The trade-off between these two possibilities has to be well studied before taken into hands. Adapting one of the *ETL* transformation algorithms may interfere with other essential parts of the framework and raise the complexity of the solution. In this case, it would be wiser to create a Scratch *Search* Algorithm and build the respective interconnections with the required parts of the frameworks. Despite the chosen algorithm, the solution's architecture will remain the same (figure 3.14). The plan is to modify EuGENia's source code to allow the generation of not only the DSL editor but also the Search Editor. The latter would have its functionality fully expressed in the plug-in to be implemented by the use of PDE.

3.3.1 *Search* Metamodel

The union of a Search Metamodel with a DSL Metamodel can reduce the complexity of the final Algorithm to be implemented. This Search Metamodel would integrate several attributes that would ease the process of *finding* specific elements. By no means that all possible search patterns will be described in this thesis as there is an enormous number of possible combinations. However, in the list below we present few benefits that this new metamodel would bring:

- **Example 1**: Each DSL Metamodel's class would inherit from the Search Metamodel a special *boolean* attribute. For instance, to *find* all the elements **A** which are not connected to elements **B**, the user would have to drop both elements to the respective editor and turn the special attribute of **B** element to false. In this way, the search would avoid all the connections between **A** and **B** and give as results all the other free elements of **A**.

- **Example** 2: Similar to the previous approach, a special attribute would be inherited from the Search Metamodel to all the classes of the DSL Metamodel. This attribute, when switched on would give as result all the instances of that element. By switching off this attribute, the *search* would only look upon the first random element that would appear in the modeling editor. The randomness search of this element could be engineered to start e.g. from the bottom-up of the model or from any other direction, requested by the user.

- *others ideas will certainly appear throughout the implementation of this project...*

The user would not have to worry about this part of the implementation since our plug-in would also take care of this integration.

3.3.2 Solution's Workflow

Activity diagrams [61] help users to understand the dynamic aspects of a system by representing the flows and decisions of the actions to be performed.
The main workflow was divided into: **editors generation** and **submission of a specific *search***. Activity Diagram 3.15 summons the process of creating the editors.
At the same time that EuGENia processes the request, the language defined by the user for the DSL has to be integrated with the Search Metamodel (described in section 3.3.1). Upon completing all these language compositions, the user receives both editors (modeling and *search*) ready to use.
The second activity diagram (figure 3.16) resumes the workflow of performing an advanced *search*. The core of this action sits on the "Solution Plug-in"because it is the bridge between the *search* editor and the GMF editor. The plug-in has the responsibility to "grab"the *search* editor's introduced pattern and look for similarities or matches within the modeling editor. In the end, the user will be able to see the highlighted results.

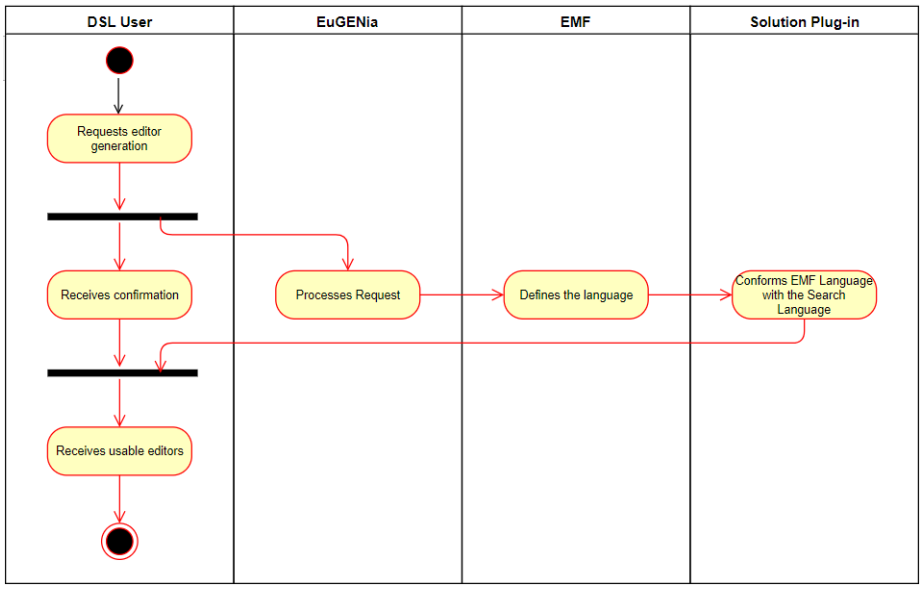

Figure 3.15: Activity Diagram representing the editors generation workflow

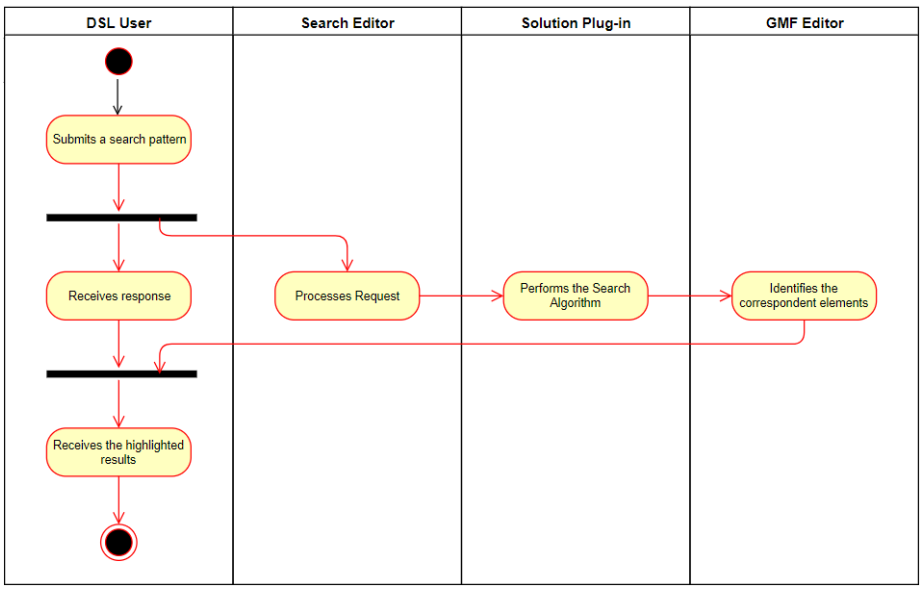

Figure 3.16: Activity Diagram representing the *search* submission workflow

Chapter 4

Work Plan

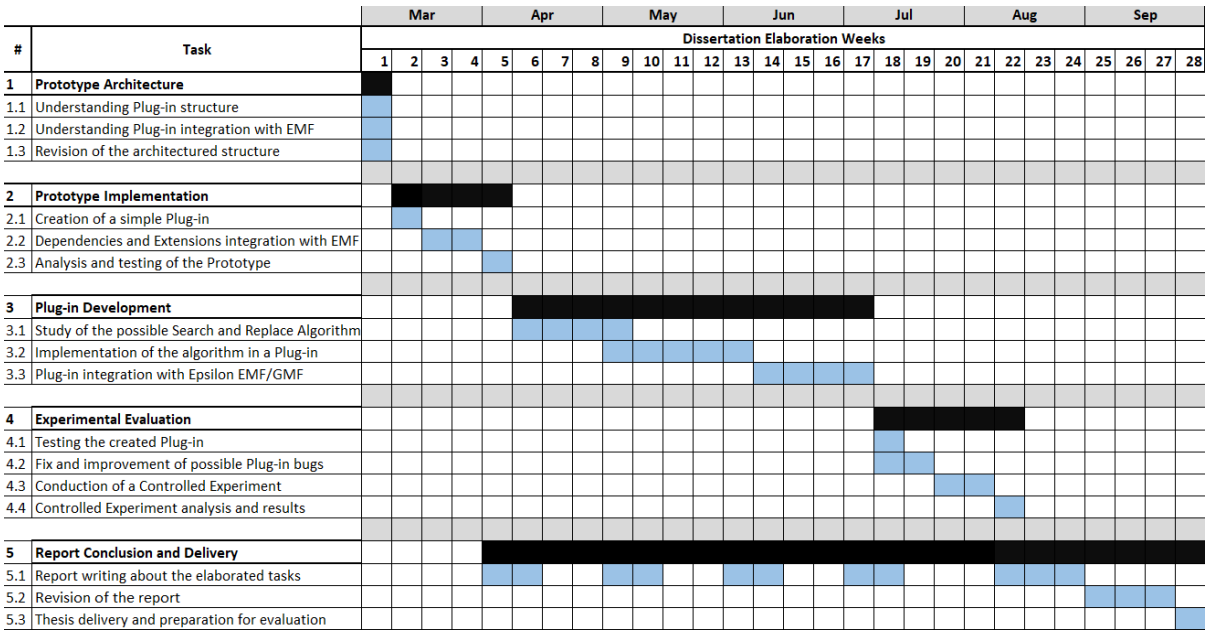

Figure 4.1: Elaboration of the dissertation's Work plan

The proposed tasks for the elaboration of the thesis are represented in table 4.1. This table is divided weekly and intends to present the management of the tasks which will be followed during the next months. The first month will focus on creating a Plug-in prototype and exploring its sensitive points. The execution of these tasks will make it easier in the future to adapt our Algorithm solution with Eclipse Epsilon holding EMF/GMF framework. The next phase (from April to June) consists in the implementation part of our solution to the problem. These three months are the most important because they will define the success and further evaluations of the created functionalities. The next phase represents the experimental evaluation and consists in testing our solution and finding possible bugs. After this brief review and correction of errors, we will conduct a controlled experiment to verify and validate the usefulness and increased usability that the implemented functionalities could bring to DSML users. The final phase will serve to complete the report by describing all of the previous tasks and to review older ones (e.g., State of the art). The last phase is to produce the thesis document report and submission.

Bibliography

[1] E. Acceleo. *https://www.eclipse.org/acceleo//*.

[2] L. T. W. Agner, I. W. Soares, P. C. Stadzisz, and J. M. SimãO. "A Brazilian Survey on UML and Model-driven Practices for Embedded Software Development." In: *J. Syst. Softw.* 86.4 (2013), pp. 997–1005. URL: http://dx.doi.org/10.1016/j.jss.2012.11.023.

[3] R. Alves. "Família de DSLs para Integrated Modular Avionics." Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2014.

[4] D. Ameller. "PhD Thesis: Non-Functional Requirements as drivers of Software Architecture Design." Universitat Politècnica de Catalunya, 2009.

[5] P. Baker, S. Loh, and F. Weil. "Model-Driven Engineering in a Large Industrial Context — Motorola Case Study." In: *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*. MoDELS'05. Springer-Verlag, 2005, pp. 476–491. ISBN: 3-540-29010-9, 978-3-540-29010-0. URL: http://dx.doi.org/10.1007/11557432_36.

[6] A. Barisic, V. Amaral, M. Goulao, and B. Barroca. "Quality in Use of Domain-specific Languages: A Case Study." In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU '11. ACM, 2011, pp. 65–72. ISBN: 978-1-4503-1024-6. URL: http://doi.acm.org/10.1145/2089155.2089170.

[7] A. Barisic, V. Amaral, and M. Goulao. "Usability Evaluation of Domain-Specific Languages." In: *Proceedings of the 2012 Eighth International Conference on the Quality of Information and Communications Technology*. QUATIC '12. IEEE Computer Society, 2012, pp. 342–347. URL: http://dx.doi.org/10.1109/QUATIC.2012.63.

[8] D. Batory, J. Thomas, and M. Sirkin. "Reengineering a Complex Application Using a Scalable Data Structure Compiler." In: *SIGSOFT Softw. Eng. Notes* 19.5 (1994), pp. 111–120. URL: http://doi.acm.org/10.1145/195274.195299.

[9] H.-U. zu Berlin. *https://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html*.

[10] J. Bézivin. "On the unification power of models." In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188. URL: https://doi.org/10.1007/s10270-005-0079-0.

[11] A. Bolour. *Notes on the Eclipse Plug-in Architecture*. 2003.

[12] M. Bone and D. R. Cloutier. *The Current State of Model Based Systems*.

[13] F. P. Brooks Jr. "No Silver Bullet Essence and Accidents of Software Engineering." In: *Computer* 20.4 (1987), pp. 10–19. ISSN: 0018-9162. URL: http://dx.doi.org/10.1109/MC.1987.1663532.

[14] J. Bézivin. "In Search of a Basic Principle for Model Driven Engineering." In: 5 (2004).

[15] J. Cabot. *https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/*.

[16] T. Catarci. "What happened when database researchers met usability." In: 25 (May 2000), pp. 177–212.

[17] T. Clark, A. Evans, P. Sammut, and J. Willans. "Applied Metamodelling: A Foundation for Language Driven Development (Third Edition)." In: (2015).

[18] K. Czarnecki and S. Helsen. "Feature-based survey of model transformation approaches." In: *IBM Systems Journal* 45.3 (2006), pp. 621–645. ISSN: 0018-8670. DOI: 10.1147/sj.453.0621.

[19] K. Czarnecki and S. Helsen. "Classification of Model Transformation Approaches." In: 2003.

[20] M. P.E. M. Dave Steinberg Fran Budinsky. *EMF - Eclipse Modeling Framework*. Second. Pearson Education, Inc., 2009. ISBN: 0-321-33188-5.

[21] E. P. in Development Environment. *http://www.eclipse.org/pde/*.

[22] E. Documentation. *http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/overview-summary.html*.

[23] E. EMF. *http://www.eclipse.org/modeling/emf/*.

[24] EMFText. *http://www.emftext.org/index.php/EMFText*.

[25] E. Epsilon. *https://www.eclipse.org/epsilon*.

[26] S. Erdweg, T. van der Storm, M. Volter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning. "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge." In: *SLE*. 2013.

[27] E. Eugenia. *https://www.eclipse.org/epsilon/doc/eugenia/*.

[28] J. marie Favre. "Megamodeling and etymology - a story of words: From MED to MDE via MODEL in five milleniums." In: *In Dagstuhl Seminar on Transformation Techniques in Software Engineering, number 05161 in DROPS 04101. IFBI*. 2005.

[29] J. Ganssle. *A trillion lines of code?*

[30] E. GMP. *http://www.eclipse.org/modeling/gmp/*.

[31] Google. *https://www.google.com/forms/about/*.

[32] D. Granada. *https://modeling-languages.com/comparing-tools-build-graphical-modeling-editors/*.

[33] E. Graphiti. *https://www.eclipse.org/graphiti/*.

[34] J. Gray and G. Karsai. "An examination of DSLs for concisely representing model traversals and transformations." In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. 2003, 10 pp.–. DOI: 10.1109/HICSS.2003.1174892.

[35] G. Génova. *Modeling and metamodeling in Model Driven Development. What is a metamodel: the OMG's metamodeling infrastructure*. 2009.

[36] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. "Empirical Assessment of MDE in Industry." In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. ACM, 2011, pp. 471–480. ISBN: 978-1-4503-0445-0. URL: http://doi.acm.org/10.1145/1985793.1985858.

[37] J. Hutchinson, J. Whittle, and M. Rouncefield. "Model-driven engineering practices in industry: social, organizational and managerial factors that lead to success or failure." In: *Science of Computer Programming* 89.Part B (2014), pp. 144–161.

[38] ISO. *International Standard Organization. ISO/IEC 9126 Quality Standards. 2004.* 2004. URL: http://www.iso.org/iso/.

[39] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. "A Software Engineering Experiment in Software Component Generation." In: *Proceedings of the 18th International Conference on Software Engineering*. ICSE '96. IEEE Computer Society, 1996, pp. 542–552. URL: http://dl.acm.org/citation.cfm?id=227726.227842.

[40] J. Kramer. "Is Abstraction the Key to Computing?" In: *Commun. ACM* 50.4 (2007), pp. 36–42. URL: http://doi.acm.org/10.1145/1232743.1232745.

[41] T. Kühne. "Matters of (Meta-) Modeling." In: *Software & Systems Modeling* 5.4 (2006), pp. 369–385. URL: https://doi.org/10.1007/s10270-006-0017-9.

[42] B. Langlois, C. elena Jitia, and E. Jouenne. *DSL Classification*.

[43] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer. "Model Transformation Intents and Their Properties." In: *Softw. Syst. Model.* 15.3 (2016), pp. 647–684. ISSN: 1619-1366. URL: http://dx.doi.org/10.1007/s10270-014-0429-x.

[44] R. M. Herndon Jr and V. A. Berzins. "The realizable benefits of a language prototyping language." In: 14 (1988), pp. 803 –809.

[45] W. Melhem and D. Glozic. *PDE Does Plug-in*. 2003.

[46] T. Mens and T. Tourwe. "A survey of software refactoring." In: *IEEE Transactions on Software Engineering* 30.2 (2004), pp. 126–139. ISSN: 0098-5589.

[47] B. Merkle. "Textual Modeling Tools: Overview and Comparison of Language Workbenches." In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 139–148. ISBN: 978-1-4503-0240-1. URL: http://doi.acm.org/10.1145/1869542.1869564.

[48] Metacase. *http://www.metacase.com/products.html*.

[49] Microsoft. *https://msdn.microsoft.com/en-us/library/bb126327.aspx*.

[50] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Tech. rep. Object Management Group (OMG), 2003.

[51] MODELWARE. "D5.3-1 Industrial ROI, Assessment, and Feedback- Master Document. Revision 2.2 (2006)." In: ().

[52] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez. "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases." In: *Empirical Software Engineering* 18.1 (2013), pp. 89–116. URL: https://doi.org/10.1007/s10664-012-9196-x.

[53] U. de Montréal. *http://www-ens.iro.umontreal.ca/ syriani/atompm/atompm.htm#people*.

[54] U. de Montréal. *http://www-ens.iro.umontreal.ca/ syriani/main.html*.

[55] P. Morville and J. Callender. *Search Patterns: Design for Discovery*. 1st. O'Reilly Media, Inc., 2010. ISBN: 0596802277, 9780596802271.

[56] Obeo. *https://www.obeodesigner.com/en/*.

[57] OMG. *http://www.omg.org/*.

[58] OMG. *http://www.omg.org/spec/MOFM2T/About-MOFM2T/*.

[59] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[60] G. Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., 1997. ISBN: 98-102288-48.

[61] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[62] D. C. Schmidt. "Guest Editor's Introduction: Model-Driven Engineering." In: *Computer* 39.2 (Feb. 2006), pp. 25–31. URL: http://dx.doi.org/10.1109/MC.2006.58.

[63] B. E.M.H.L.K.E.V.G. W. Markus Voelter with Sebastian Benz Christian Dietrich. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. 2013.

[64] E. Seidewitz. "What models mean." In: 20 (2003), pp. 26 –32.

[65] B. Selic. "What Will It Take? A View on Adoption of Model-based Methods in Practice." In: *Softw. Syst. Model.* 11.4 (2012), pp. 513–526. ISSN: 1619-1366. URL: http://dx.doi.org/10.1007/s10270-012-0261-0.

[66] S. Sendall and W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development." In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.

[67] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN: 0470025700.

[68] H. Storrle. "Large scale modeling efforts: a survey on challenges and ¨ best practices." In: *W. Hasselbring (Ed.), Proceedings of the IASTED International Conference on Software Engineering (IASTED-SE'07), Acta Press2007*, pp. 382–389.

[69] H. Störrle. "VMQL: A visual language for ad-hoc model querying." In: 22 (Feb. 2011), pp. 3–29.

[70] M. V. Sven Efftinge. *oAW xText: A framework for textual DSLs*. 2006.

[71] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, V. Mierlo, and H. Ergin. *AToMPM: A Web-based Modeling Environment*.

[72] E. Syriani, H. Vangheluwe, and B. LaShomb. "T-Core: a framework for custom-built model transformation engines." In: *Software & Systems Modeling* 14.3 (2015), pp. 1215–1243. ISSN: 1619-1374. URL: https://doi.org/10.1007/s10270-013-0370-4.

[73] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio. "Relevance, Benefits, and Problems of Software Modelling and Model Driven techniques-A Survey in the Italian Industry." In: *J. Syst. Softw.* 86.8 (2013), pp. 2110–2126. URL: http://dx.doi.org/10.1016/j.jss.2013.03.084.

[74] Q. Treasury. *Presenting survey results - Report writing*.

[75] E. E. Tutorial. *https://www.eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial/*.

[76] *Unified Modeling Language Infrastructure, Version 2.0*. Tech. rep. Object Management Group (OMG), 2004.

[77] V. University. *http://www.isis.vanderbilt.edu/Projects/gme/*.

[78] E. Visser. "WebDSL: A Case Study in Domain-Specific Language Engineering." In: *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*. Ed. by R. Lammel, J. Visser, and J. Saraiva. Springer Berlin Heidelberg, 2008, pp. 291–373. ISBN: 978-3-540-88643-3. URL: https://doi.org/10.1007/978-3-540-88643-3_7.

[79] T. Weigert. "Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems." In: *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing -Vol 1 (SUTC'06) - Volume 01*. SUTC '06. IEEE Computer Society, 2006, pp. 208–217. ISBN: 0-7695-2553-9-01. URL: https://doi.org/10.1109/SUTC.2006.106.

[80] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. "Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?" In: *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*. Springer-Verlag New York, Inc., 2013, pp. 1–17. ISBN: 978-3-642-41532-6. URL: http://dx.doi.org/10.1007/978-3-642-41533-3_1.

[81] E. Xpand. *https://eclipse.org/modeling/m2t/?project=xpand*.

[82] E. Xtend. *http://www.eclipse.org/xtend/*.

[83] E. Xtext. *https://www.eclipse.org/Xtext/*.

Appendix

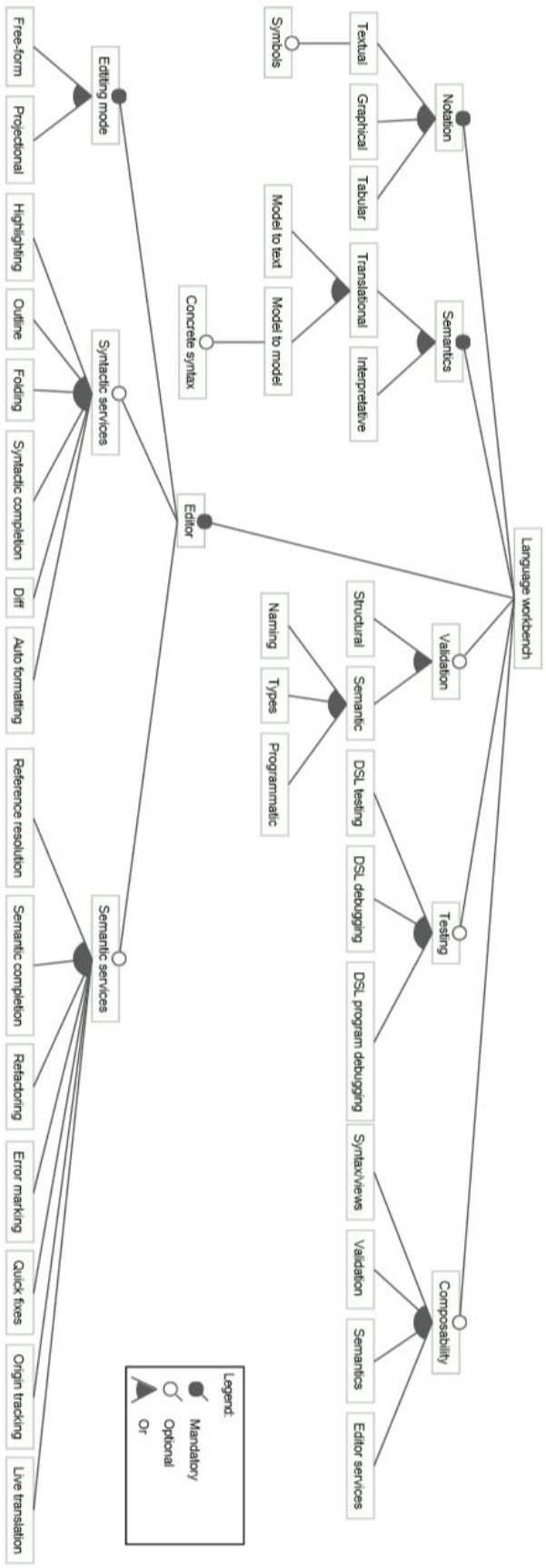

Figure A.1: Language Metamodeling Workbench Feature Model

ANNEX I

Survey research on Usability aspects of Model-Driven tools: Model find and replace techniques

I.1 Introduction

Model-Driven Development (MDD) is a paradigm for writing and implementing computer programs quickly, effectively and at minimum cost and at a higher level of abstraction than General Purpose Languages[65]. UML, IFML, Domain-Specific Languages (DSL) are some of the languages that are model-driven related. By using these, the application's code can be generated based on the designed model (e.g. Class Diagram, Activity Diagram in case of UML, etc.), saving precious time to the system developers.

Our current work is centered on building solutions to help the current DSL modelling workbenches to empower its developers and end-users with better means to achieve a higher level of productivity.

I.2 Background and objectives

This survey[1] was conducted to better understand the preference of Software Engineers nowadays - either if they prefer a model or a centric approach - and to understand if the development of a new functionality that lacks in any of the DSL workbenches would help to bring more audience to the model-driven paradigm.

[1] https://goo.gl/forms/eLCd0Jjqo8pKw8c22

I.3 Survey method

The survey was implemented online using Google Forms [31] and comprises mostly single choice questions. It was judged to take no longer than 5 minutes to complete. Even though this survey was preferably intended for Software Engineers who had used Model-Driven technologies in some moment of their academic or professional career (preferably), most of the applicants were informatic students from NOVA University. Also, some employees and teachers were invited to answer this survey. Since this questionnaire was anonymous it was not possible to know whom exactly answered it, either if it was a student, employee or teacher.

I.4 Survey results

I.4.1 Univariate analysis

Represent the type of analysis that is understandable from directly looking at the answers without relating any questions among themselves. All of the questions below pretend to answer problems that MDD is facing at the current time. These are being deeply studied by many computer scientists that have published numerous articles (e.g. [37, 65, 80]), with the intention of moving forward in this subject.

I.4.1.1 Question 1: How often do you use Modeling-Driven tools?

As we can observe in figure I.1 represented in the Survey Appendix section (I.7), 68% of the applicants rarely use Model-Driven tools to develop; 16% claim to use them occasionally and 16% never used them.

I.4.1.2 Question 2: Have you ever developed a Model-Driven application for commercial purposes?

As we can observe in figure I.2, 84% of the applicants claim to never have developed a commercial Model-Driven application before. The small number of 16% have some experience before with this kind of methodologies.

I.4.1.3 Question 3: Do you find the Model-Driven Development approach adequate for software development?"

As we can observe in figure I.3, 56% of the applicants moderately think that MDD approach is adequate for software development; 24% are indifferent to it; 17% definitely think that MDD is adequate for software development; 4% think that MDD is moderately not adequate; 4% think that definitely, it's not worth it.

I.4.1.4 Question 4: Which Integrated Development Environments for developing Model-Driven applications are the best in your opinion?

As we can observe in figure I.4, 40% of the applicants think that Microsoft DSL Tool is the best Integrated Development Environment (IDE) for MDD; other 40% think that Eclipse/Eugenia suits better their needs as an IDE; 24% don't even know any IDE; 12% thinks EMFText is the best; 4% prefer ATomPM.

I.4.1.5 Question 5: In which paradigm do you feel more comfortable to develop?

As we can observe in figure I.5, 68% of the applicants prefer a Code Centric approach against the other 32% who prefer a Model-Centric style.

I.4.1.6 Question 6: In how many commercial MDD projects have you participated in as a software developer?

As we can observe in figure I.6, most of the applicants (82,6)% never participated in a commercial MDD project; 13% participated in just one project and 4,3% participated in three projects.

I.4.1.7 Question 7: Do you think that an advanced model find and replace functionality in DLS workbenches could ease the process of developing a complex DSL using diagrammatic/visual DSLs?

As we can observe in figure I.7, 52% of the applicants feel indifferent in whether an advanced model find and replace functionality could ease the process of developing a complex DSL; 24% definitely think that it can be helpful; 20% moderately think that it could ease the process; 4% doesn't have any hope that by improving the usability of a DSL workbench, a step forward in MDD could be taken.

I.4.2 Bi-variate analysis

Represent the type of analysis that is understandable from directly looking at the answers and relating two or more questions among themselves.

I.4.2.1 Question 1 & 3

100% of the people that occasionally used Modeling-Driven tools (fig.I.1) think that MDD is an adequate approach for software development; Approximately 60% of the people that rarely used still think that MDD is a good approach whether the rest are indifferent to it or just don't think it is a good approach; The applicants that never used it all have a different opinion.

	Q.1-Occasionally	Q.1-Rarely	Q.1-Never
Q.3-Definitely yes	25%	6%	25%
Q.3-Moderately yes	75%	58%	25%
Q.3-Indifferent	0%	30%	25%
Q.3-Moderately no	0%	6%	25%
Q.3-Definitely no	0%	0%	0%

Table I.1: Survey - Question 1 & 3

I.4.2.2 Question 2 & 3

100% of the applicants that have developed an MDD application for commercial purposes (fig. I.2) think that it is definitely or moderately adequate to use these kind of methodologies for software development (fig.I.3). This analysis shows the acknowledged importance of MDD by the people that used it already, recognizing its value and potential; The people that never developed a commercial MDD application are divided in its benefits (62% think would be adequate against 38% that don't).

	Q.2-Yes	Q.2-No
Q.3-Definitely yes	25%	10%
Q.3-Moderately yes	100%	52%
Q.3-Indifferent	0%	28%
Q.3-Moderately no	0%	5%
Q.3-Definitely no	0%	5%

Table I.2: Survey - Question 2 & 3

I.4.2.3 Question 3 & 5

100% of the applicants that feel more comfortable using a Model-Centric approach (fig. I.5) are absolutely or moderately sure that an MDD approach is adequate for software development (fig.I.3). Approximately 60% of the people who feel more comfortable with code-centric approach still think that it is adequate to follow an MDD style as a software development process. This bi-variate analysis shows the importance of making people understand how really does a Model-Driven Development approach works and how they can get better results from it. The first step is to try the shift from a Code Centric approach to a Model-Centric and embrace it for the necessary time in order to understand it correctly since many people are still indifferent to this methodology process.

I.4.2.4 Question 6 & 7

As we can see from the table I.4 the higher the number of projects made, the higher it is the recognition by the applicants that a new tool would be useful to them in the context of a complex DSL. The people who have used MDD before, are aware of its potential and that it can improve quality in developing complex DSLs. The applicants who participated

	Q.5-Model Centric	Q.5-Code Centric
Q.3-Definitely yes	25%	6%
Q.3-Moderately yes	75%	53%
Q.3-Indifferent	0%	35%
Q.3-Moderately no	0%	6%
Q.3-Definitely no	0%	0%

Table I.3: Survey - Question 3 & 5

in zero MDD projects are more indifferent (57%) to the use of this methodology and are not so interested in the improvement of the DSL workbenches usability, perhaps because they lack on its possible potential.

	Q.6-Zero projects	Q.6-One project	Q.6-Three Projects
Q.7-Definitely yes	19%	33%	100%
Q.7-Moderately yes	19%	33%	0%
Q.7-Indifferent	57%	33%	0%
Q.7-Moderately no	0%	0%	0%
Q.7-Definitely no	5%	0%	0%

Table I.4: Survey - Question 6 & 7

I.5 Threats to validity

Maybe some of the participants weren't quite familiar with the concept of Model-Driven Development or simply had lack of experience in it. Also, most of the population were students that were more into a Code Centric approach rather than a Model approach. One of the (not mandatory) questions were not presented in this report because it was not so well understood by the participants and got only a few answers, most of them asking the purpose of the same, which means the sentence could have been better constructed.

I.6 Conclusion

From the different types of analysis, it's noticeable a lack of knowledge from a large group of individuals upon the model-driven methodology. By analyzing the survey answers and relations among them, it is possible to understand that people who had used MDD before are aware of the advantages that this kind of methodology can bring. The people who are familiar with it are the ones that also acknowledge that it is necessary to improve the usability of these tools by creating new functionalities - e.g. a model find and replace technique -. Since the number of participants was quite low - only 25 -, a deeper understanding from this survey can't be lightly taken. More answers are required to be possible to retain a deeper study and more certainties about this specific subject. Although the

number of participants is not as high as expected, it is clear that Model-Driven Development is a methodology with a great potential but that needs to be more shared with the Software Engineering community.

I.7 Survey Appendix

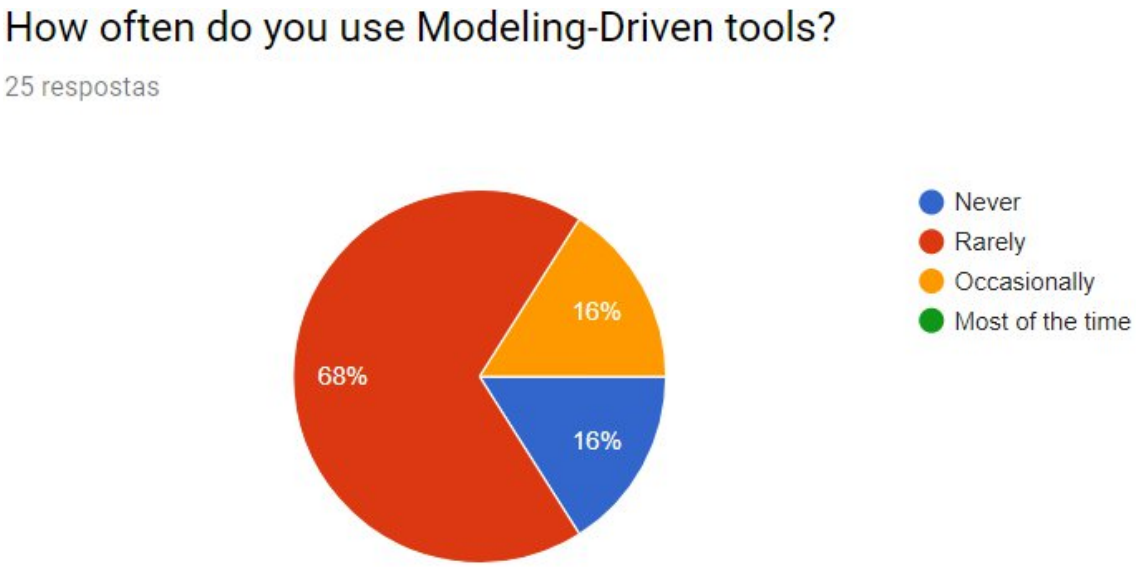

Figure I.1: Survey - Question 1

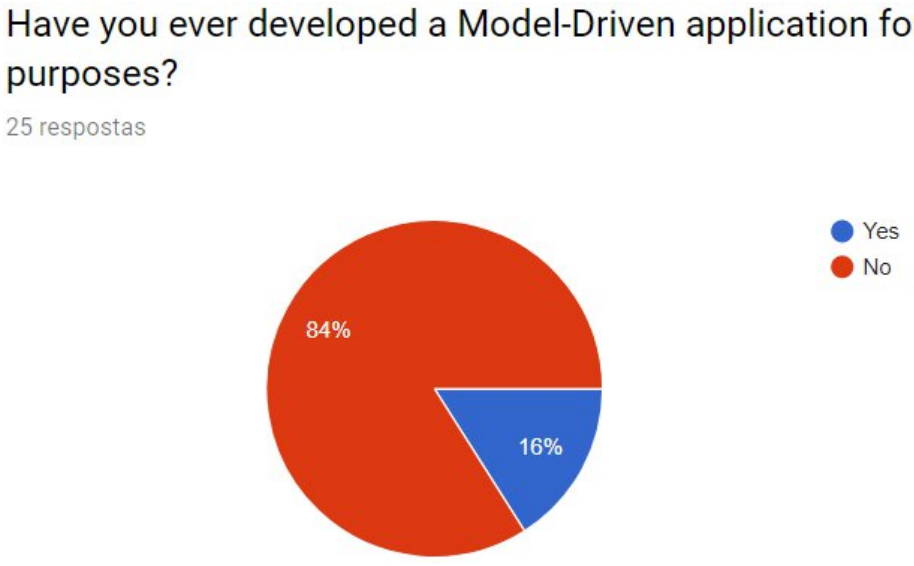

Figure I.2: Survey - Question 2

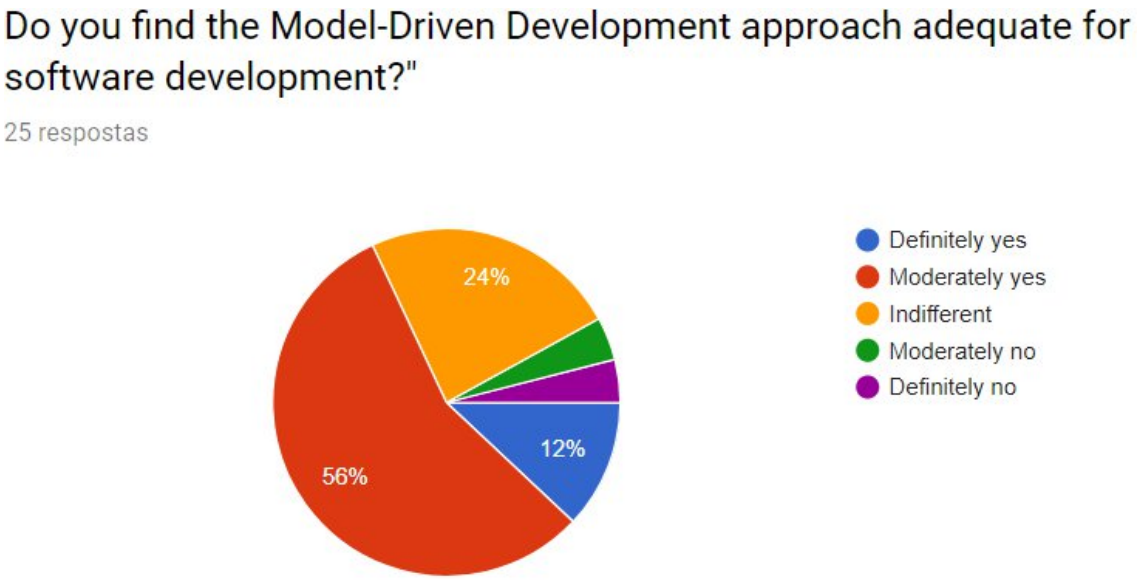

Figure I.3: Survey - Question 3

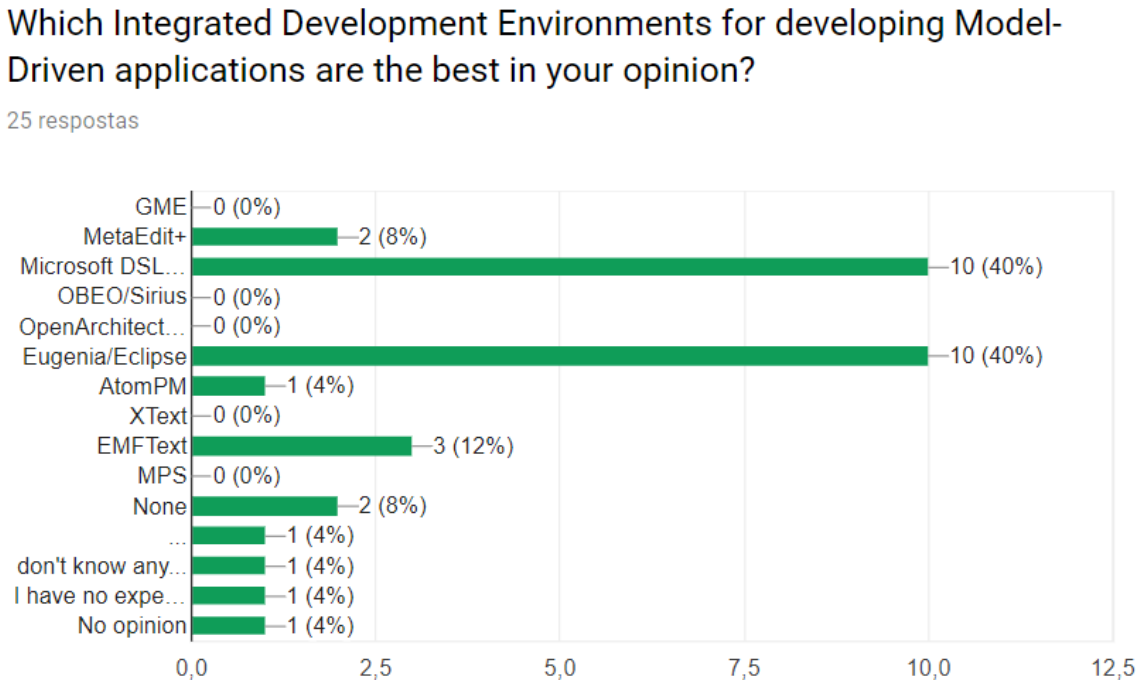

Figure I.4: Survey - Question 4

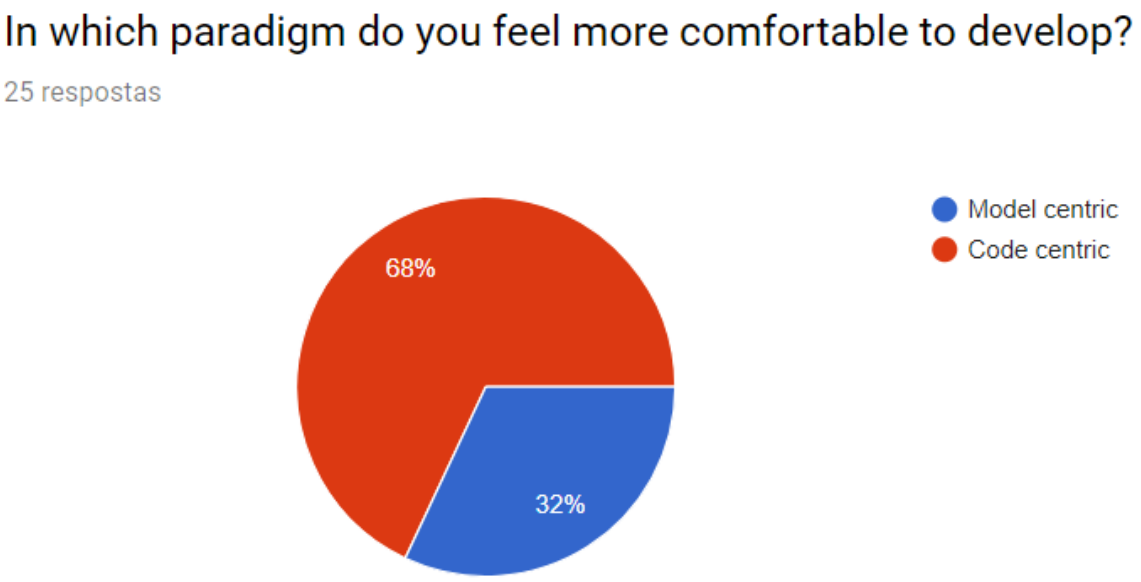

Figure I.5: Survey - Question 5

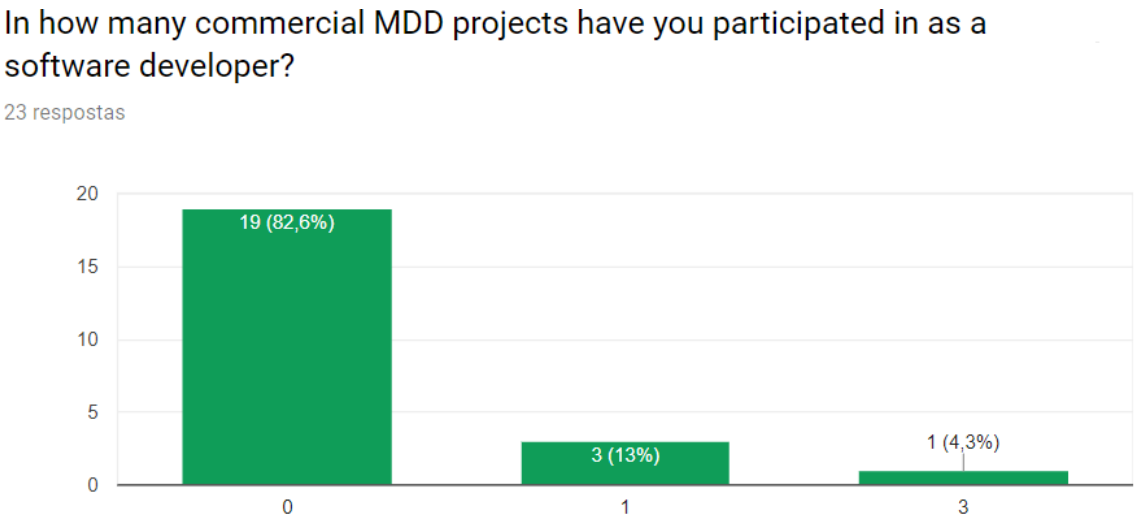

Figure I.6: Survey - Question 6

Do you think that an advanced model find and replace functionality in DLS workbenches could ease the process of developing a complex DSL using diagrammatic/visual DSLs?

25 respostas

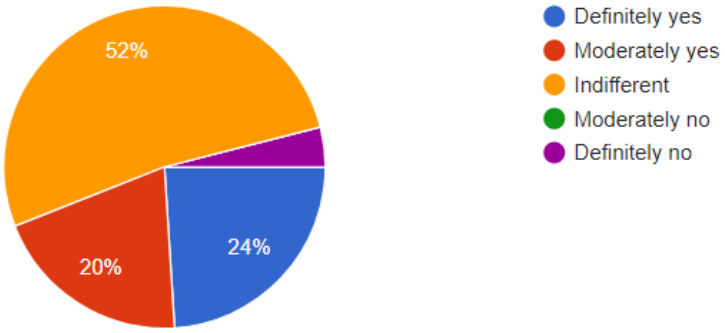

Figure I.7: Survey - Question 7