Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-MSC-2010-01

Masterarbeit

im Studiengang "Angewandte Informatik"

An Evaluation of Model Transformation Languages for UML Quality Engineering

Lucas Andreas Schubert

am Institut für

Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

March 1, 2010

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (5 51) 39-17 42010
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den March 1, 2010

Master's Thesis

An Evaluation of Model Transformation Languages for UML Quality Engineering

Lucas Andreas Schubert

March 1, 2010

Supervised by Prof. Dr. Grabowski
Software Engineering for Distributed Systems Group
Institute for Informatics
Georg-August-Universität Göttingen

Acknowledgments

First of all, I would like to thank my family and fiancée for all their love and support. A special thank to my tutor Benjamin Zeiss, which always answered my questions and e-mails, even on weekends or late night. I would also like to thank Prof. Dr. Jens Grabowski for giving me the opportunity of writing this thesis. Further thanks go to Dr. Markus Grumann, that make it possible for me to work during my studies, by giving me more flexibility in my working scheduling, without this support, it would be very difficult to continue studies. Finally I would like to thank all my friends here and in Brazil, colleagues, and everyone who helped me directly or indirectly in my studies.

Abstract

Detecting modeling errors in the first stages of a software development process can spare time and money. Software quality engineering is a field of computer science for evaluating the quality of software and providing mechanisms to ensure software quality. This thesis evaluates the transformation languages ATLAS Transformation Language (ATL), Epsilon Transformation Language (ETL), Query/View/Transformation (QVT), and Xtend by analyzing their characteristics in relation to the International Organization for Standardization (ISO) 9126 standard, a language characteristics taxonomy proposed by Czarnecki and Helsen, and their applicability for calculating metrics and detecting bad smells in Unified Modeling Language (UML) models. A case study has been used to evaluate the transformation languages in the task of executing a Model to Model (M2M) transformation for calculating metrics and detecting bad smells. All four transformation languages are suitable for UML quality engineering, however there are differences, such as performance issues or tooling characteristics that should be taken into consideration.

Keywords: Unified Modeling Language, Model to Model Transformations, Metrics, Bad Smells, ATL, ETL, QVT, Xtend

Contents

1. Introduction **1**
1.1. Related Work 2
1.2. Contributions 3
1.3. Thesis Structure 4

2. Foundations **5**
2.1. Model Driven Engineering (MDE) 5
2.2. Model to Model Transformation (M2M) 6
2.3. Unified Modeling Language (UML) 8
2.4. Software Metrics 10
2.5. Bad Smells 11
2.6. Refactoring 11
2.7. International Organization for Standardization (ISO) 9126 Standard 12
2.8. A Classification for Evaluating Model Transformation Languages 14
2.9. Languages and Technologies 17

3. Case Study **30**
3.1. Bad Smell Detection and Metrics Calculation Metamodel 31
3.2. The Example Model 33
3.3. Bad Smell Detection 37
3.4. Model Metrics 38

4. Case Study Implementation **40**
4.1. Overview 40
4.2. ATLAS Transformation Language (ATL) 41
4.3. Epsilon Transformation Language (ETL) 46
4.4. Operational QVT Language 49
4.5. Xtend 54

5. Language Evaluations **59**
5.1. Analyzed Subjects of the Transformation Language 59
5.2. Evaluation Schema 60
5.3. ATLAS Transformation Language (ATL) 62

5.4. Epsilon Transformation Language (ETL) 74
5.5. Operational QVT 85
5.6. Xtend 96

6. Conclusion **106**
6.1. Outlook 108

List of Tables **108**

List of Figures **109**

List of Listings **110**

A. Acronyms **113**

Bibliography **117**

1. Introduction

The software industry has become one of the largest industries on the planet, and many of today's most successful businesses are software production companies or offer services in the software field [142]. Software are used for a large variety of tasks, from the firmware running in digital clocks to software used to predict natural disasters. Such software can be developed using software development methodologies. Many of these methodologies provide mechanisms for enhancing the software quality.

One of these methodologies is Model Driven Engineering (MDE), where models are used to describe and define static structures and dynamic behaviors of an application, and play a key role in the development process of an application. These models are usually based on standard specifications, such as the Object Management Group (OMG) Unified Modeling Language (UML) [130, 131] specification. UML is a general-purpose visual modeling language that is used to specify, visualize, and document the artifact of software systems [139]. Software specified by models provide many features, such as a representation of the application that can be understood by software developers familiar with the UML specification, generating the source code of an application using automated tools based on model transformations, detecting bad smells and anti-patterns [83] in the model before implementing the application, calculating metrics in the model for predicting development time or executing refactoring [99] in the model before starting to code. Model transformation plays a key role in many tools used in MDE, such as model refactoring tools, metric calculation tools executed over models, and bad smell detection tools.

This work evaluates four different transformation languages based on the characteristics specified in the ISO 9126 standard [106], on the language characteristic evaluation proposed by Czarnecki and Helsen [89], and their capability for handling UML quality engineering tasks such as calculating metrics and detecting bad smells.

The four transformation languages evaluated in this thesis are the ATLAS Transformation Language (ATL) [7], the Epsilon Transformation Language (ETL) [31], the Query/View/Transformation (QVT) Operational Mapping Language (OML) [128], and Xtend [47]. These languages are used in tasks such as transforming a model into another model, this process is known as M2M transformation. The ATL transformation language, available in the Eclipse Modeling Tools distribution [29], is a hybrid transformation language that supports imperative and declarative implementations. The ETL transformation language is also a hybrid transformation language, such as ATL, and is part of the Extensible Platform of Integrated Languages for mOdel maNagement (Epsilon) project

that is hosted as an Eclipse project. Epsilon provides a set of different languages that are task specific, in which ETL is used for M2M transformations. The imperative language QVT OML is one of the transformation languages from the OMG QVT specification. Since it is a specification, there are many different implementations available. The imperative transformation language Xtend is the Model to Model (M2M) transformation language used in openArchitectureWare (oAW). Since oAW version 5, Xtend it is hosted as part of the Xpand project in the Eclipse platform.

These four transformation languages are evaluated in a case study with M2M transformations related to UML quality engineering. The case study consists of calculating five metrics, detecting three bad smells in the source model, and generating a target model containing all metrics calculated for each element analyzed, and the bad smells detected. This case study is implemented in each transformation language evaluated to analyze in a more practical way its characteristics, features, and drawbacks.

The International Organization for Standardization (ISO) 9126 standard defines six quality characteristics that are used to evaluate software products. The transformation languages implementations, and their tools are evaluated based on this ISO characteristics. Another evaluation approach that was included in this thesis is the continuity evaluation for analyzing the future perspective of the transformation language.

The characteristics related to the transformation languages themselves are evaluated based on the Czarnecki and Helsen [89] classification. This methodology classifies model transformation approaches based on their characteristics. By using this article for evaluating the languages, it is possible to find important characteristics that are relevant in M2M transformation processes related to quality engineering.

The evaluation conclusion is based on three different sources of information. A practical experience using the transformation languages in implementing the case study, by analyzing the transformation languages ISO 9126 characteristics, and in evaluating the languages characteristics using the classification defined in [89]. Combining this information, it was possible to make a clearer evaluation of the transformation languages related to UML quality engineering in the tasks of detecting bad smells, and metric calculations.

1.1. Related Work

This thesis evaluates transformation languages in M2M transformations, focusing on UML quality engineering. Several other studies are related to the evaluation of transformation languages or related to quality engineering.

The study from Czarnecki and Helsen [89] defines a classification for evaluating transformation approaches. Several transformation approaches have been analyzed, and a categorization based on this approaches was proposed. Using this categorization it is possible to evaluate important features related to model transformation approaches.

The work from Jouault and Kurlev [110] analyzes the interoperability between ATL, and other M2M transformation languages such as QVT, *Tefkat* [61], VIsual Automated model TRAnsformations (VIATRA2) [69], Graph Rewrite And Transformation (GReAT) [38], Attributed Graph Grammar System (AGG) [64]. This work first analyzes the transformation languages individually based on the classification of [89] and other classifications referenced in the study. Then the actual interoperability between ATL and each of the transformation languages are evaluated individually. In [109], the same authors also evaluate ATL individually based on [88].

A master thesis from Huber [105] evaluates four different transformation languages using a criteria catalog based on [88]. The four evaluated languages are ATL, SmartQVT [57], Kermeta [40], and ModelMorf [44]. First an overview of the classification used is given, and then the languages are analyzed. Then based on this classification, each transformation language is evaluated in detail.

A discussion about the existing work in the field of quality assessment and improvement of UML models is described by Jalbani et. al. in [107].

The article from Tsantalis and Chatzigeorgiou [146] proposes a detailed methodology for identifying *Move Method* refactoring opportunities. A methodology for executing the refactoring is described, such as the preconditions for executing the refactoring. This methodology is also implemented as an Eclipse plugin. At the end the proposed methodology is evaluated.

1.2. Contributions

The contributions of this thesis are the following:

- The case study executes a M2M transformation in which it calculates five software metrics in the source model, and based on this metrics, three bad smell types are detected. The target model containing the metrics and bad smells entries conform to a metamodel specified in this thesis. This metamodel has been developed so that metric calculation results are stored in such a way that it is possible to calculate the bad smells without having to analyze the source model. This metamodel has also been developed to be used as second source model containing the bad smells that must be refactored in refactoring processes.

- The same case study is implemented in the four different evaluated transformation languages.

- The transformation languages have been evaluated based on an evaluation schema, and an analysis of the transformation language characteristics based on [89]. The

evaluation schema contains topics that are relevant to specific ISO 9126 characteristics. It is divided into two groups, one with topics related to the implementation, and the other with topics related to the research. The ISO 9126 characteristics, and the transformation language classification are evaluated individually taking in consideration the use of the language for UML quality engineering.

- A new classification named *continuity* has been included in this thesis to evaluate the transformation language future perspective. This classification is also evaluated in the evaluation schema.

1.3. Thesis Structure

This thesis is structurated as following. In chapter 2 the foundations and important background information relevant for this master thesis are briefly described. This chapter gives an overview of the MDE methodology, the Model to Model Transformation (M2M) approach, Unified Modeling Language (UML) specification, Metrics, Bad Smells, Refactoring, the ISO 9126 standard, the model transformation classification approach, and the main technologies and transformation languages related to this work.

Chapter 3 describes the case study, and it is subdivided in one section that describes the Bad Smell Detection and Metrics Calculation Metamodel (BSDMC_MM) metamodel used in the M2M transformation as the target metamodel. Another section that describes the source model example used in the case study. One section contains a description of the metrics calculations implemented in this case study, and another section describes the bad smells detection implementations.

The case study implementation is described in chapter 4. First an overview about the implementation, such as common information to all implementations is given. The sections ATLAS Transformation Language (ATL), Epsilon Transformation Language (ETL), Operational Query/View/Transformation (QVT), and Xtend describes the case study implementation from each transformation language evaluated.

In chapter 5 the transformation languages are evaluated based on the ISO 9126 standard, and also analyzed based language characteristics classification proposed on [89]. In chapter 6 a summary of the conclusions and an outlook of the work is presented.

2. Foundations

This chapter provides the foundations and the theoretical basis of this thesis. An overview of MDE, and its development cycle is described in section 2.1. Section 2.2 gives an introduction of M2M transformations and briefly describes different M2M transformation approaches. Section 2.3 introduces UML, giving an overview of its diagram structure and architecture. An introduction of software metrics is presented in section 2.4, giving an overview about metrics, and listing a few object oriented software metric approaches. In section 2.5 is the definition of bad smells briefly described and section 2.6 briefly presents the refactoring process. The ISO 9126 used in the evaluation schema in the discussion chapter is described in section 2.7. The model transformation classification proposed by Czarnecki and Helsen [89] is presented in section 2.8. The technologies and transformation languages used in the case study are introduced in section 2.9.

2.1. Model Driven Engineering (MDE)

The idea promoted by MDE is to use models at different levels of abstraction for developing systems [97]. This models can be used for documenting the software, source code generation based on models, detecting quality issues from modeled applications using on automated tools, and enhancing the software portability and interoperability. There are also other terms such as Model Driven-Software Development (MDSD), Model Driven Development (MDD), and Model Driven Architecture (MDA) [125], but they do not always imply the same methods. Compared to MDSD, MDA tends to be more restrictive, focusing on UML-based modeling languages [142]. The main objectives of MDA, as described in [104], are portability, system integration and interoperability, efficient software development, domain orientated, and capability of displaying specific knowledge in the model.

Figure 2.1 describes how software source code is generated from a model using MDA. The Platform Independent Model (PIM), is a model that does not contain platform specific or library specific information modeled. The information necessary to transform the PIM into a Platform Specific Model (PSM) are contained in the Platform Description Model (PDM). The Transformation Description Model (TDM) also contains information that is relevant to the process of generating a PSM. The PSM is a platform specific model, where information of a specific programming language and libraries are contained. Based on the

PSM model is the source code generated. This source code generated is usually composed by classes, methods and packages. Additional information about MDE can be found in [104, 134, 142].

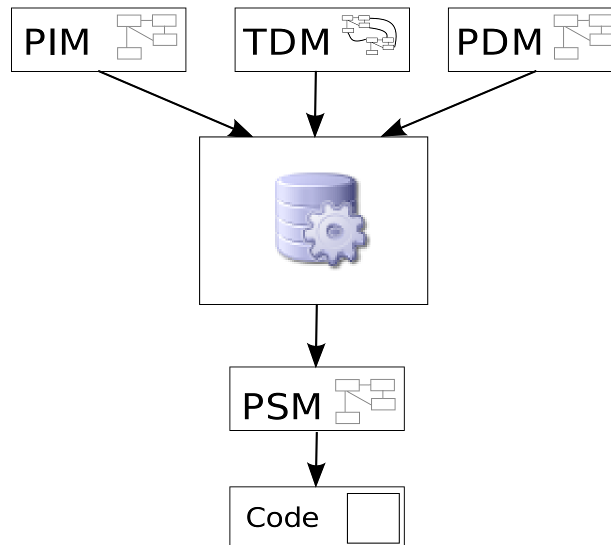

Figure 2.1.: Code Generation Process in MDA

2.2. Model to Model Transformation (M2M)

In M2M, one model is transformed into another model. We can use model transformations, for example, to convert one UML model into a model based on a different metamodel. To perform this transformation, diverse technologies such as ATL, Eclipse Modeling Framework (EMF) Model Query [30], Epsilon, *Xtend* and QVT may be used. In general, a transformation has as input a source model, the source metamodel, the target model in the case that they are different, and the target model instance in the case that the model is not recreated, but only modified. Each transformation language has its own way of processing model elements. ATL, for example, uses rules to map and execute transformations, such as matched rules that are called automatically by the transformation process. The QVT OML transformation language has one input function. Inside this function is declared mappings between the target and source models, and these are called based on the implementations order. As a transformation result, a new or modified target model instance is generated.

There are many different transformation approaches for performing M2M transforma-

tions. The M2M transformation approaches can be classified as described below according to [89].

- Direct manipulation: This approach provides an internal model representation and also an Application Programming Interface (API) to manipulate it, such as Java Metadata Interface (JMI). A programming language such as Java [22] is used to program it. In this approach the user must program the transformation rules, tracing, rules scheduling and the other facilities related to the transformation. It is also usually implemented as an object oriented framework.

- Structure-driven: This approach is divided in two phases, in the first phase the hierarchical structures of the target model are created, and in the second phase the attributes and references in the target model are set. The users have to write the transformation rules, while the framework is responsible for the rules scheduling. One example of this approach is the framework provided by OptimalJ [18].

- Operational: This category groups the M2M transformation approaches that are similar to the direct manipulation approach, but offer a more dedicated support for model transformation. One example of extending the support for model transformations according to [89] is by extending the metamodeling formalism, with facilities for expressing computations. Examples of transformation approaches in these categories are QVT Operational [128], XMF-Mosaic executable Meta Object Facility (MOF) [71], and Kermeta [40].

- Template-based: In this approach, according to [89], model templates are models with embedded metacode that compute the variable parts of the resulting template instances. These templates are usually expressed in the concrete syntax of the target language. This metacode may be referenced as annotations on model elements. A concrete example of this approach is described in [87].

- Relational: Declarative approaches based on mathematical relations are grouped in this category. They can be seen as a form of constraint solving. Examples of relational approaches are QVT Relations [128], Model Transformation Framework (MTF) [2], Tefkat [61], and Atlas Model Weaver (AMW) [11].

- Graph-transformation-based: This transformation approaches are based on theoretical work on graph transformations. They operate on typed, attributed, labeled graphs, and its rules have a Left Hand Side (LHS) and a Right Hand Side (RHS) graph patters. Examples of this approach are A Tool for Multi-formalism Meta-Modelling (AtoM3) [15], From UML to Java And Back Again (Fujaba) [33], and MOdel transformation LAnguage (MOLA) [112].

- Hybrid: This category consists of a combination of the approaches described above. Examples are ATL, Yet Another Transformation Language (YATL) [133], and QVT.
- Others: According to [89], the following two approaches are mentioned for completeness, and are Stylesheet Language Transformation (XSLT) [72] and the application of metaprogramming to model transformation.

2.3. Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a family of graphical notations, backed by single metamodel, that help in describing and designing software systems, particularly software systems built using the Object Oriented (OO) style [98]. The main goal in specifying UML is the creation of a general purpose modeling language that all users can understand. The OMG adopted the UML as a standard in November 1997. It was born from the unification of many OO graphical languages such as Object Modeling Technique (OMT), Booch method, and Objectory between the late 1980s and the early 1990s.

UML has descriptive rules, so that there is not an official way of modeling a software structure or behavior in UML. These rules of how a specific part of the software should be modeled could be defined by the group of software engineers working on that specific project. There are recommended notations that have been accepted by convention, but the use of them are not obligatory. It is also important to remark, that because of the UML complexity, models specified based on the recommendations can be open to multiple implementations.

The software engineer and developers are free to model one specific part of the application, using the diagrams that they think will better describe what they want to document. But in cases of a M2M transformation or a source code is generated based on one model, the entire model must be consistent to a specification sufficient to generate the source code or to execute M2M transformation.

UML is divided in structural and behavior diagrams. The structural diagrams are used to model the software structure, such as components, the relation between components, classes, methods, etc. The behavior diagrams are used to model the software behavior, such as a specific sequence of methods calls or the modeling of a state machine. Figure 2.2 shows the UML available diagrams.

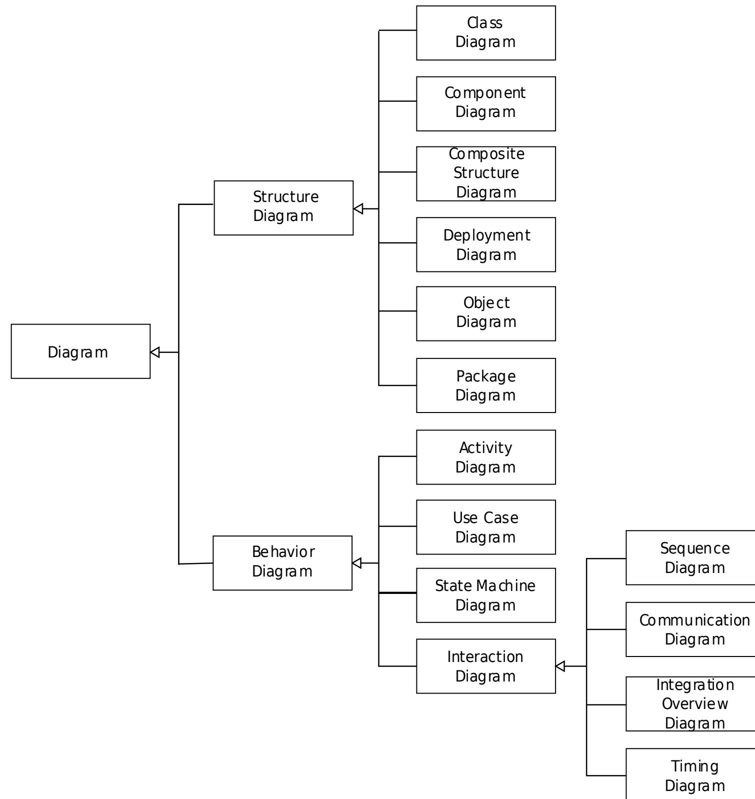

Figure 2.2.: UML Diagrams

Class Diagrams are recommended in cases that classes, features, and relationships should be modeled. *Component Diagrams* are recommended in cases that the modeling purpose is to model the structure and the connections of components. The *Composite Structure Diagrams* are recommended in modeling the runtime decomposition of a class cases. In the case that the modeling purpose is to model the deployment of artifacts to nodes, the *Deployment Diagram* is recommended. The *Object Diagrams* are unofficially adopted in UML 1, and it is recommended to use in cases that instances configurations should be modeled in one specific time in the application's runtime. The *Package Diagram* has also been unofficially adopted in the UML 1 specification, and is recommended when the purpose of the modeling is to model the dependency between packages.

Activity Diagrams are recommended in cases that the modeling purpose is to model procedural and parallel behavior. The *Use Case Diagram* is recommended to model how the users interact with the modeled system. The recommended diagram to model how events change an object over its life is the *State Machine Diagram*. A *Sequence Diagram* is recom-

mended in cases that the modeling purpose is to model the interaction between objects with emphasis on sequence. *Communication Diagrams* are recommended for modeling the interaction between objects, but emphasis on links. The *Interaction Overview Diagram*, is a mix between the *Sequence Diagram* and the *Activity Diagram*. *Timing Diagrams* is recommended in cases that the modeling purpose is to model the interaction between objects but with emphasis in timing, in real time applications for example.

The UML architecture is composed by four layers, and shown in Figure 2.3. The *M3* is the first layer, and all other layers are based on it. It is called MOF, and it is considered a language that is used to model other models or metamodels, and can also be used to model itself. The MOF is also an OMG specification, and can be found in [126]. On top of the *M3* layer is the *M2* layer, where the Unified Modeling Language version 2 (UML2) metamodel is located. This metamodel is an instance of MOF. On top of the *M2* layer is the *M1* layer that is an *User Model* that is an instance of the UML2 metamodel. One example of an *User Model*, is the example model used in the case study on chapter 3. The *M0* layer where the model is instanced based on the *M1* layer.

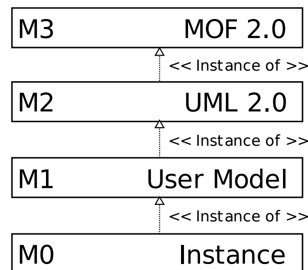

Figure 2.3.: UML Metamodeling

2.4. Software Metrics

Software metrics deals with the measurement of software and the applications development processes. These metrics can be subdivided in groups such as size, complexity, quality, and object oriented metrics. Software metrics can be used to predict and track software projects, identify bad smells in the software source code, or calculate and detect reliability aspects of software. The use of software metrics can enhance the quality of software, as also be used in supporting decisions related to its development process.

An example is the Logical Lines of Code (LLOC) metric used to represent the actual amount of source code lines, not considering empty or commented lines. This metric can be used to measure the size of the application, or to give an overview how is the source code distributed between the files. Such information is valuable in supporting the decision

making of for example predicting the time needed to refactor one entire file.

Metrics can also be calculated in models, not only in the source code. Calculating metrics, in the first stages of the model development of one application, can save time and money. It is possible to detect and remove bad smells in the modeling phase, avoiding that these bad smells propagate to the source code. There are also specific metrics for object oriented models and implementations. Examples of object oriented metrics are *Weighted Methods per Class*, *Response for a Class*, *Lack of Cohesion of Methods*, *Coupling Between Object Classes*, *Depth of Inheritance Tree*, and *Number of Children*. Detailed information about these metrics is available in [138].

The startup of metrics measuring in object oriented design, according to [77] occurred in 1991 by two pioneers, Shyam R. Chidamber and Chris F. Kemeler [84]. In 1994 they published a work [85] about the Metrics for Object-Oriented Software Engineering (MOOSE) metric, also known as C.K. metrics. Other approaches came after, such as the Metrics for Object-Oriented Design (MOOD) [95], Quality Model for Object-Oriented Design (QMOOD) [80], Formal Library for Aiding Metrics Extraction (FLAME) [82], and others. Additional references to object oriented metrics can be found in [77, 81, 140].

2.5. Bad Smells

Bad smells can be found in the source code or in models. In source codes, bad smells are dubious implementations in the source code that may generate an error, or it does not conforms to a specific metrics calculated value that indicates good implementation practices. Bad smells can be found not only on procedural languages but also in object oriented languages. Examples of bad smells in the source code are a method with too many lines of code, a method that has too many parameters, a class with too many parameters, and etc.

In MDA, the source code is one of the last parts in the software development process. Bad smells can also be found in models, such as UML, or Ecore models. If bad smells are detected and removed in this phase of the development process, they do not propagate to the source code. A reference about bad smells can be found in [99].

2.6. Refactoring

According to [99], refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing the observable behavior. The refactoring process can be executed in the source code level or in the model level. In the software development process, it is common to adapt and modify the software during its life cycle. This modifications may be necessary to correct a bug, to include new func-

tionalities, to adapt one part of the software structure and/or behavior to a specific *Design Pattern* [100], or to remove bad smells. In all this cases refactoring processes can be used.

Fowler proposes a catalog of refactoring in [99], in which each refactoring has a *name*, *summary*, a *motivation* that describes why a refactoring should be done and in what circumstances it should not be done, the *mechanics* that describes step-by-step how to perform the refactoring, and an *example* about how the specific refactoring works. Other references related to refactoring are [91, 122, 145, 146]

2.7. International Organization for Standardization (ISO) 9126 Standard

ISO 9126 [106] quality model standard is an international standard used for evaluating the software quality. This ISO norm is divided in quality models for external and internal quality, and quality-in-use. This thesis evaluates the transformation languages based on the external and internal quality model. Figure 2.4 shows the quality model for external and internal quality.

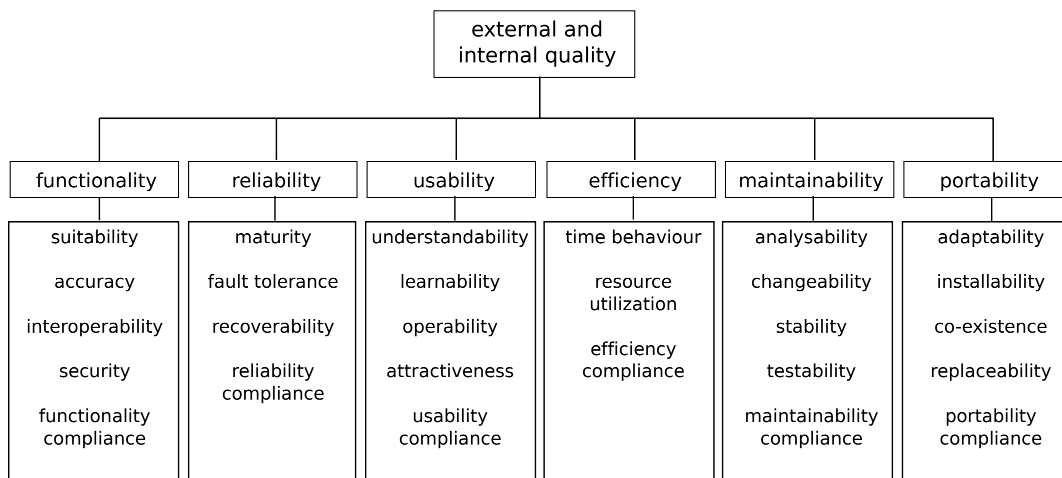

Figure 2.4.: ISO 9126 quality model for external and internal quality.

The main categories *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability* defined in this standard are described below.

2.7.1. Functionality

This category describes if the technology is suitable by providing all needed functions to perform one specific task, accurate to generate the wanted results, interoperable between systems, secure in protecting the data information, and provides functionality compliance that defines that a technology compliance to other standards, conventions, regulations, and also similar prescriptions relating to the functionality.

2.7.2. Reliability

The capability of the software product to maintain a specified level of efficiency when used under specified conditions [106]. Important subcategories are the maturity, which consists on the capability of avoiding failure based on faults in the software. Fault tolerance, which consists of keeping a specified level of efficiency after a fault occurred. Recoverability, that consists of re-establishing the same level of efficiency after one fault occurred. And reliability compliance that defines that technology compliance to other standards, conventions, regulations, and also similar prescriptions relating to the reliability.

2.7.3. Usability

The usability category is related to the level of difficulty to understand the technology, the level of difficulty to learn how the technology works, the level of operability in the case of how the user can control the technology, the technology attractiveness representing features that makes it compelling for the user, and also usability compliance to other standards, conventions, regulations, and also regulations and style guides related to the usability.

2.7.4. Efficiency

The efficiency is related to time behavior, such as the time needed to execute one specific process. It is also related to the resource utilization, and also that the resources that are used are appropriate to the task executed.

2.7.5. Maintainability

The maintainability is an important characteristic, and represents topics such as the technology analyzability for identifying causes of failure or to identify one specific implementation part. The changeability that refers to the capability of changing the implementation, stability in the case that an implementation change does not generate unexpected effects, testability referring to the capability of testing and validating the implementation, and that the capability to adhere to standards, and conventions relating to maintainability.

2.7.6. Portability

Portability is the capacity that one technology can be transferred from one environment to another, and is subdivided in the adaptability, that refers to the capacity of adapting the software to other environment without having to perform large changes. Installability is the capacity of the software of being installed in one specific environment. The capacity of the software to co-exist with other independent technology in a common environment is known as co-existence. The capability of replacing the technology with another one that has the same purpose is known as replaceability, and at last the portability compliance, that refers that the technology adhere to standards, and conventions relating to portability.

2.8. A Classification for Evaluating Model Transformation Languages

There are many approaches to perform M2M transformations in models. Each transformation approach has advantages and disadvantages depending on its characteristics. In [89], those approaches are collected and divided in categories that make it possible to categorize most model transformation approaches. It divides the model transformations in two major categories, the Model to Text (M2T) and M2M transformations. The M2T transformations are subdivided in the *Visitor-based*, and *Template-based* approaches, while the M2M transformations are subdivided in *Direct manipulation*, *Structure-driven*, *Operational*, *Template-Based*, *Relational*, *Graph-transformation-based*, *Hybrid*, and other approaches. Figure 2.5, and the following subsections shows and describes the top level feature diagram from the classifications.

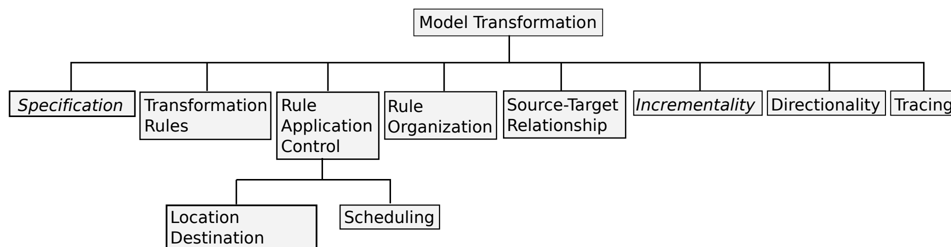

Figure 2.5.: Top Level feature classification

2.8.1. Specification

In some approaches there are dedicated specification mechanism, such as the use of *pre* and *post* conditions that can be expressed in the form of Object Constraint Language (OCL)

expressions or functions that must be validated between target and source model. These specifications describe relations, therefore, in general they are not executable. According to [89], the QVT-Partners submission distinguished between relations as potentially non executable specifications of their executable implementation. This classification is optional.

2.8.2. Transformation Rules

In this classification, transformation rules are considered as a broad term for describing the smallest unit of transformations, so that not only rules are considered as transformation rules, but also functions and procedures that somehow executes one transformation. This transformation rules are subdivided in *domains*, that is the part of the rule responsible for accessing one of the models that are used in the rule transformation. A *domain* is again subdivided in *domain language*, *static mode*, *dynamic mode restriction*, *body*, and *typing*. In this subdivision is defined for example in how many *domain languages* the rule operates, how are the domains declared, if they are implicitly or explicitly declared, such as if a specific domain is defined as *in*, *out*, or *inout*, or if the rule *body* supports variables for example. Other classifications from the transformation rules are also the *syntactic separation*, *multidirectionality*, *application controls*, *intermediate structures*, *parameterization*, *Reflection*, and *Aspects*.

2.8.3. Rule Application Control

The rule application control is divided in two aspects, the *location determination* that is related to strategy for locating the matches of the rules that are going to be executed in the source model. This strategy can be *deterministic*, *nondeterministic*, or *interactive*. One example of a *deterministic* strategy is a depth-first search in the source model, an example of a *nondeterministic* strategy is when a rule is applied to a random element in the source model. The *nondeterministic* strategies are again subdivided in *concurrent*, and *one-point*. One *interactive* strategy approach is when the transformation process allows the user to interact with the transformation to determine the location where the rule is going to be applied.

The rule *scheduling* is related to the order that individual rules are applied, and they are subdivided in four main areas that are : *form*, *rule selection*, *rule interaction*, and *phasing*. The *form* refers to the way scheduling can be expressed implicitly or explicitly. The *rule selection* defines how the rule is selected and it can be composed by *explicit condition*, *nondeterministic*, *conflict resolution*, and *interactive*. The *rule interaction* refers to rules mechanisms for interactions, and it can be *recursion*, *looping*, or *fixpoint interaction*. *Phasing* refer to the possibility of dividing a transformation process in many phases, so that the rules that are going to be executed in each phase can be defined. The *rule interaction*, and *phasing* are optional characteristics.

2.8.4. Rule Organization

The rule organization is related to how the rules are organized and structured, and they were divided in three areas, the *modularity mechanism* that defines how de rules can be organized, such as spreading the rules in modules. The *reuse mechanism* refer to the use of rules in conjunction, or a way to reuse rules, examples of rules reuse are rule inheritance, inheritance between modules, and rules extensions. The *organizational structure* refer to how the rules are organized, rules can be attached to elements in the source model, or in the target model for example. The characteristics *modularity mechanisms*, and *reuse mechanisms* are optional.

2.8.5. Source-Target Relationship

Source-Target relationship refers to how the transformation handles the models. A transformation may create a new target model, edit a target model by performing an *update*, or an *in-place* transformation. In the case of an *update*, it is subdivided in *destructive* and *extension only*. There are approaches that allow the creation of new target models, and also edit them, and there are approaches that only allow a generation of a new target or editing an existing target.

2.8.6. Incrementality

The incrementality is related to the ability to update target models based on a change in the source model. This topic is subdivided in three groups, *target incrementality*, *source incrementality*, and *preservation of user edits in the target*. This classification is also considered optional. The *target incrementality* refers to the ability of changing a target model, based on changes performed in the source model. This feature is also referred as *change propagation* in the QVT final adopted specification [89]. The *source incrementality* refers to minimizing the source that must be reexamined in a transformation when a source models is modified. And *preservation of user edits in the target* refers to the ability of rerun a transformation over a target model, keeping the modifications made by the user in the target model.

2.8.7. Directionality

The directionality is related to the transformation direction, one transformation can be *unidirectional* in the case that the target model is created or modified based on a source model. It can also be *multidirectional*, in cases where the transformations are performed in two directions, such as in models synchronization. This kind of transformation can be achieved by defining multidirectional rules, or by defining several unidirectional rules, one in each direction.

2.8.8. Tracing

Tracing can be understood as the runtime footprint of the transformation process. These traces are useful for analyzing the connections between the target and the source model in the transformation, observing what target element was created based on what source element, it can also be useful for performing impact analysis, determining the target of a transformation as model synchronization, model based debugger, and also in debugging the model transformation process.

2.9. Languages and Technologies

This section describes an overview of the technologies and transformation languages that are relevant in this thesis.

2.9.1. ATLAS Transformation Language (ATL)

ATLAS Transformation Language (ATL) is the ATLAS INRIA & LINA research group answer to the *OMG MOF / QVT Request For Proposal (RFP)* [78]. It is considered a hybrid transformation language, since it contains a mixture of imperative and declarative constructs. It is encouraged to use the declarative style of specifying transformations. But in some cases it is necessary to use the imperative features of the language. ATL performs a one way transformation, i.e., it is only possible to read the input model, and write the output model.

The ATL Development tools are composed by the ATL transformation engine and the ATL Integrated Development Environment (IDE), i.e., is composed by an editor, compiler, and debugger. The output model can be generated in the XML Metadata Interchange (XMI) [127] or Kernel Meta Meta Model (KM3) [79] format. The standard library available is composed by the primitive types, collection types and its operations and they are based on the OCL 2.0 [132] standard library. There is also a Virtual Machine (VM) called *Regular VM*, and another named *EMF VM*. Additional information about the library and its implementations for both Virtual Machines is available in [8].

The engine is responsible for compiling the ATL code into a specialized byte code, that is executed by the ATL VM. The ATL editor supports syntax highlighting, error reporting, and also an outline view. The ATL compiler is automatically called for each ATL file in each ATL project during the Eclipse build process [108].

The source code is divided in header, helpers and rules sections. The module name, models, and metamodel variables are defined in the header section. The keywords that are used in this section are the `module` keyword that defines the module name. The `create` keyword that introduces the target model declaration, and the `from` keyword introduces the source model declaration (Listing 2.1).

```
1 module Proceedings2Articles;
2 create OUT : Articles from IN : Proceedings;
```

Listing 2.1: ATL `header` example

It is possible to import other libraries to be used in the transformation. The `uses` keyword followed by the name of the library is used to include it in the transformation.

Helpers are subroutines that are used to avoid code redundancy [76]. They are used to define global variables and functions that can be specified as a helper with or without context. The `helper` functions are OCL expressions that also provides recursion. The keyword `helper` defines a new helper `context`. The context is set as `metamodel!element`. The `helper` syntax is shown in Listing 2.2.

```
1 helper [context]? def : attribute_name : return_type = exp;
```

Listing 2.2: ATL `header` syntax

The `self` variable inside the helper represents the context element. After the keyword `def` the helper name is defined. One helper example is shown in Listing 2.3.

```
1 helper context Proceedings!Paper def : getAuthors : String =
2   self−>collect(e | e.author)−>asSet()−>iterate(vAuthorName; vAuthorsStr : String = '' |
3                                                 vAuthorsStr +
4                                                 if vAuthorsStr = ''
5                                                 then vAuthorName
6                                                 else ', ' + vAuthorName
7                                                 endif);
```

Listing 2.3: ATL `header` example

The rules are the central part of the transformation, rules describes how output elements are generated based on the input elements. Both depend on each corresponding model. One module may contain more then one rule. But the rule's name must be unique inside the module. The listing below describes the rule syntax structure.

```
1  rule rule_name {
2    from
3      in_var : in_type [(
4            condition
5      )]?
6    [using { var_1 : var_type_1 = init_exp_1;
7          ...
8          var_n : var_type_n = init_exp_n;
9    }]?
10   to
11     out_var_1 : out_type_1 (
12       bindings_1
13     ),
14     out_var_2 : distinct out_type_2 foreach (e in collection) (
15       bindings_2
16     ),
17     ...
18     out_var_n : out_type_n (
```

```
19         bindings_n
20     )
21   [do {
22         statement
23   ]?}
```

Listing 2.4: ATL `rule` syntax

After the keyword `rule` the rule name is defined. Each rule has the mandatory blocks `from` and `to`, and the optional blocks `using` and `do`. The `from` block refers to the element from the source model that must match to the type element `in_type`, and as an optional block, also satisfy the *condition* statement. The `using` is also an optional block used to initialize variables. The `to` block corresponds to the target pattern. It contains a number of target elements that are going to be instantiated. The `do` block is optional and it enables the use of a sequence of ATL imperative statements, that are executed after the target elements have been instantiated and initialized. The listing below shows one rule example.

```
1  rule Paper2Articles {
2    from
3      b : Proceedings!Paper (
4              b.getPages() > 15
5      )
6    to
7      out : Articles!Article(
8          title <− b.title,
9          authors <− b.getAuthors()
10     )
11 }
```

Listing 2.5: ATL `rule` example

In the example above (Listing 2.5), the `to` block will only be executed if source element is from the `Paper` type that has more than 15 pages. In this example, for each `Paper` element from the source model that satisfies the filter, one element from the type `Article` will be created in the target model.

2.9.2. Epsilon

Epsilon is a platform for building consistent and interoperable task-specific languages for model management tasks such as model transformation, code generation, model comparison, merging, refactoring and validation [118]. It is a component that belongs to the Generative Modeling Technologies (GMT) research incubator. The development tools available in [31] for Epsilon are EuGENia, Exeed, ModeLink, Workflow, and Concordance. The Epsilon project is not available in the standard *Eclipse Modeling Tools*, so it must be installed separately.

Epsilon is not considered as a transformation language, but it is a set of interoperable task-specific languages for model management. This set of interoperable task-specific lan-

guages consists of seven languages, and each language is used for one specific model management task, such as model transformation, code generation, model comparison, merging, refactoring and validation. The Epsilon languages:

- Epsilon Object Language (EOL): The EOL language provides a set of common model management facilities that are used by one of the task specific languages. It can be considered as kind of a template language for the other ones, since all other languages are built on top of EOL. It is also possible to execute it as a standalone model management language for automating tasks that does not belong to one of the specialized language domain.

- Epsilon Validation Language (EVL): EVL is the language used for model validation in Epsilon. Features such as a detailed user feedback, constraint dependency management, semi-automatic transactional inconsistency resolution, and access to multiple models of diverse metamodels are available in EVL. There are similarities between EVL and OCL, but EVL is not based on the OCL specification.

- Epsilon Transformation Language (ETL): The language used for model transformation in Epsilon is ETL. It is a M2M transformation language that supports the transformation of many input into many output models, it is rule based, its possible to modify the input and output model, provides declarative rules with imperative block, and other features.

- Epsilon Comparison Language (ECL): ECL is a language used to compare models. It is a hybrid, rule-based language that can be used to compare homogeneous and heterogeneous models. It can also be used to establish correspondences between models, can export comparison results to Epsilon Merging Language (EML) or to a custom model/format and other features.

- Epsilon Merging Language (EML): Such as ECL, EML is a hybrid, rule based language for merging homogeneous and heterogeneous models, can export the merge trace to a custom model/format, provides declarative rules in imperative blocks, and other features.

- Epsilon Wizard Language (EWL): EWL is a language tailored to interactive in-place model transformations on user-selected model elements (unlike ETL which operates in a batch mode)[118]. Features such as executing wizards within EMF and Graphical Modeling Framework (GMF) editors, define guards in wizards, undo and redo effects of wizards on the model are available.

- Epsilon Generation Language (EGL): It is a M2T template based language for generating textual artifacts such as code, and documentation for example. Features such

as decouple content from destination, template calls with parameters from other templates, calling of sub-templates, mixture of generated and hand-written code are available.

The EOL uses the keyword `operation` to define functions in the source code. These operations can be specified as with or without context. The use of context improves the understandability. It also provides the simple and executable types of *annotations*. It is also possible to define pre and post conditions in user defined operations.

The example below (Listing 2.6), shows an `operation` that contains pre and post conditions.

```
1  $pre aValue > 0
2  $post _result > self
3  operation Integer addValue(aValue : Integer) : Integer {
4    return self + aValue;
5  }
```

Listing 2.6: EOL pre and post condition in operations example

The code in lines 1 and 2 defines the pre and post conditions to execute the operation in line 3. The core library implements the functions that are similar to OCL for working with collection and maps, types, strings, and model element types. Detailed information about the core library is available in [118]. The operations of the type `Any` was inspired by the *OCL* implementation of `OclAny`. Operations such as `isTypeOf()`, `isKindOf()`, `asSequence()`, `asSet()` and many others are very similar to the OCL. Not only the operations used in types, but also the use of collections, and its operations are very similar to the OCL specification. The use of `If`, `While`, and `For` statements are also provided in the *EOL* language. Other feature that EOL also provides is the use of transactions. EOL implementations can also be called directly by the workflow.

On top of EOL is ETL, it inherits all features provided by EOL. Such as described above, ETL is a rule based language, and is the central part of the transformation. Each ETL file may contain more than one rule defined.

The Listing 2.7 shows the ETL rule abstract syntax. A rule can be `abstract`, `lazy`, and `primary`, the name of the rule is defined after the keyword `rule`, and must be unique. The `transform` keyword defines the source element type and the `to` keyword defines the target element type. A rule can also have more than one target element defined. The `extends` keyword is optional, and specifies a list of rules that it extends. The `guard` keyword is also optional and can define a condition to execute the rule's body block.

```
1  (@abstract)?
2  (@lazy)?
3  (@primary)?
4  rule rule_name
5    transform in_var:in_type
6    to out_var_1:out_type_1,
7        ...,
```

```
8        out_var_n:out_type_n
9     [extends rule_name_1,
10           ...
11           rule_name_n]?{
12    [guard (:expression | {statements})]?
13    statements
14    }
```

Listing 2.7: ETL `rule` syntax

Listing 2.8 shows one example of a rule implementation used in the case study.

```
1 rule CalculateMetricsInClasses
2    transform vs: Source!Class
3    to vMetricNOC : QueryResult!MetricResult,
4       vMetricCBO : QueryResult!MetricResult
5    {
6      fResult.resultsList.add(CalculateMetricNOC(vMetricNOC,vs));
7      fResult.resultsList.add(CalculateMetricCBO(vMetricCBO,vs));
8      fResult.resultsCount := fResult.resultsCount + 2;
9 }
```

Listing 2.8: ETL `rule` example

In this example, for each `Class` element in the source model, are two `MetricResult` elements from the target model created (Lines 3 and 4). These elements are inserted in the `resultsList`, which is a set of elements that contains all calculated metrics and bad smells detected in the target model. Each function, `CalculateMetricNOC` and `CalculateMetricCBO`, create a return element that contains the information related to the calculated metric.

It is possible in an ETL transformation, to define `pre` and `post` blocks for the entire transformation, so that before all rules are called, the `pre` block is executed, and after all rules has been called, the `post` block is executed. More information about the Epsilon project can be found in [118], and [115].

2.9.3. Query/View/Transformation (QVT)

Query/View/Transformation (QVT) is the new standard for the implementation of model transformations based on the MOF and OCL standards [104]. The QVT specification defines imperative and declarative transformation languages. The declarative languages are known as *relation* and *core*, while the imperative language is known as *operational mappings*.

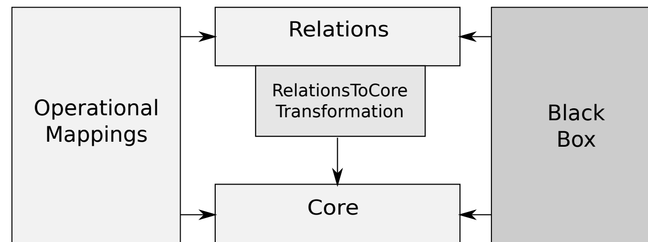

Figure 2.6.: QVT metamodels relationships

The *core* language is a declarative language, which is defined at a lower level of abstraction, and is a minimal extension to Essential Meta Object Facility (EMOF) and OCL. In this language, the users are not responsible for the creation and deletion of objects, and the traces are not automatically generated. In the *core* language, the users must define the transformation rules and trace information as a MOF metamodel. According to [128], it can only support pattern matching over a flat set of variables by evaluating the conditions over those variables against a set of models.

The *relations* language is a declarative and user-friendly transformation language mainly built around the concept of object-patterns. It supports a complex object pattern matching. The relationships between the source and target models are defined by the user. The user is not responsible for the creation and destruction of objects. It also has trace functionalities. An example extracted from [142] is shown in Listing 2.9.

```
1  transformation alma2db(alma: AlmaMM, db: DbMM)
2  top relation EntityToTable {
3    prefix, name : String;
4    checkonly domain alma entity:Entity {
5      name = eName
6    };
7    enforce domain db table:Table {
8      name = eName
9    };
10   where {
11     prefix = '';
12     RecordToColumns(entity,table,prefix);
13   }
14 }
```

Listing 2.9: QVT *relations* source code example

In the example above, the name followed by the keyword `transformation` defines the transformation name, all models used in the transformation are passed as parameters. The transformation direction is defined in the QVT *relations* transformation call. There are two types of relations, the top-level relations and non-top-relations. The execution of the transformation requires that all top-level relations hold. Non-top-level relations are required to hold only in cases where they are invoked directly or transitively from the `where` clause

from a different relation. After the keyword `relation` is defined the relations name, in the example above, `EntityToTable`. In this transformation for each element from type `Entity` contained in the model `alma`, provided they at least define a property `name`, which has to be bound to the variable `eName`. The similar occurs with the model `db`. The `checkonly` and `enforce` keywords constrains how the transformation will occur in a given direction. In a transformation direction from `alma` to `db`, the `EntityToTable` element will match all `Entity` elements from the source model and will check if a corresponding element exists in the `db` target model that has the same `name` as in the source model. If no element was found in the target model, the QVT engine generates one new entry of a `table` element in `db` model.

The imperative variant is provided by Operational Mapping Language (OML). It is intended to provide an imperative alternative, which can be invoked from the Relations and Core Languages in a "Blackbox" manner [103]. It can be used in two ways, or as a transformation executed exclusively using OML, that is also known as *operational transformation*, or the hybrid variant, that is the use of OML combined with the declarative languages. Other characteristic is that it is possible to use OCL statements in the OML language. The *Black Box* shown in Figure 2.6 is a mechanism of incorporating other implementations using different programming languages in the process.

An OML module is basically divided in header and implementation parts. In the header section are included the declaration from the models used in the transformation. The keyword `modeltype` is used to declare each model. It is followed by a name that is going to be used as reference to the model in the file, the keyword `uses` and the actual model location. An example is shown below. (Listing 2.10)

```
1 modeltype Target uses
2   'http://www.eclipse.org/emf/2002/Ecore';
3 modeltype Source uses
4   'http://www.eclipse.org/uml2/2.1.0/UML';
```

Listing 2.10: OML header example

The second section is the implementation section that is defined by the keyword `transformation`. In the example below (Listing 2.11), two variables are passed as parameters. The keywords `in` and `out` defines the transformations direction. It is also possible to use the keyword `inout` in the case that the output model should not be cleared and rewritten. The `main` function is the transformations entry point. The "`->`" string represents the mapping, the `Class2EClass()` function is executed every time one `Class` instance in the source model is found.

```
1 transformation uml2rdb(in uml:Source, out ecore:Target){
2
3   -- The main entry point of the transformation
4   main(){
5       -- Standard model element access and mapping invocation
6       uml.objects()[Class]-> map Class2EClass();
```

```
7
8   }
9 }
```

Listing 2.11: OML example

The `mapping` operations provide a fundamental behavior in the transformation, the Listing 2.12 below shows a general syntax of a mapping operation.

```
1  mapping [direction]? context::mapping_name
2    ([direction_n parameter_name_1 : parameter_type_1,
3      ...
4      direction_n parameter_name_n : parameter_type_n]?) : return_type_1, ..., return_type_n
5    [when { statements }]?
6    [where { statements }]?
7  {
8    [init { statements }]?
9    (population { statements }) | statements
10   [end { statements }]?
11 }
```

Listing 2.12: OML `mapping` general syntax

The `mapping` keyword represents the mapping operation, followed by an optional keyword to specify its direction. The `context` represents the mapping context, followed by the mapping name. Each mapping can contain none or many parameters. After the double points the return types are defined. A mapping can return more than one element, the keyword `result` is used to reference the return object, tuple, or multiple results. Before the body containing the `init`, `population`, and `end` is executed, the two optional blocks `when` and `where` are executed. The `when` clause acts either as guard, or as pre-condition, while the `where` clause acts as post-condition. The body block contains the `init` block, that is optional and it is executed before the other two blocks. It can be used to initialize variables. In the `population` block is a section to write the main code. It is not necessary to set the `population` block, only in some special cases it must be set. The `end` block is executed at last, and should contain all implementation that should be done at the end of the transformation.

The Listing 2.13 shows a mapping used in the case study implementation. The `CalculateMetricMP` mapping has one input parameter. The `mapping` body is only executed if the `where` clause is satisfied. In the `Init` block the return element is initialized. In the `population` block from lines 15 to 17 is called another mapping and the `value` property from the initialized `result` object is set.

```
1  mapping Source::Operation::CalculateMetricMP(
2      in aBehaviorDiagramsList : OrderedSet(Behavior)) : BSDMC_MM::MetricResult
3    when {self.name <> ''}{
4    init {
5      result := object BSDMC_MM::MetricResult{
6        id := 3;
7        text := 'Calculate Method Polymorphism (MP)';
```

```
8        targetObject := self.name;
9        targetObjectType := self.metaClassName();
10       parentObject := self.owner.oclAsType(Class).getQualifiedName();
11       parentObjectType := self.owner.oclAsType(Class).metaClassName();
12       type := MetricsTypes::mtMethodPolymorphism;
13     };
14   }
15   self.CollectOperationsHasBehaviorDiagram()−>xcollect(e |
16              result.metricResultItem += result.map CreateMetricItem(e));
17   value := result.metricResultItem−>size();
18 }
```

Listing 2.13: OML `mapping` example

It is also important to remark that OML also support the use of `query` and `helper`. According the QVT specification, a `query` is a special kind of `helper` operation. But unlike a `query`, a `helper` may have side effects on the parameters passed into it.

A nice analogy to the functions of each language is described in [128], in which the three languages are compared to the Java Virtual Machine (JVM), so that the *Core Language* is such as the Java byte code. The *Relation Language* plays the role of the Java language and the standard transformation from Relations to Core is such as the specification of a Java compiler, which produces the code.

The Standard library in the QVT OML specification is built on top of the OCL Standard Library, and adds pre-defined types to the *M1* layer and are specific from QVT.

2.9.4. Xpand/Xtend

Xpand is a project hosted in the Eclipse platform, and it was previously part of the oAW version 4 framework. The new oAW version 5 was subdivided in Eclipse projects, and it is hosted in the Eclipse website, while its older version is available in [47]. The openArchitectureWare (oAW) version 5 available in Eclipse was subdivided in the *Xtext* project, that is a framework for the development of Domain Specific Language (DSL), and programming languages. The Xpand, Xtend, and Check languages those are available as the Xpand project in Eclipse, which is a framework for MDSD, and the Modeling Workflow Engine (MWE) project, that is a framework for integrating and orchestrating model processing frameworks.

The Xpand project is used for M2T and M2M transformations in MDSD. The transformation language used for M2T is Xpand, an example of a M2T transformation is the source code generation based on a model. The Xtend language is a language that is used for M2M transformations, such as performing a refactoring in a model, while the *Check* language is used for analyzing constraints.

All this languages are built on top of the *Expressions Language*, so that three languages inherit characteristics from the base language. All three languages can operate on the same models, and metamodels. Features available in the *expressions language* are:

- It is possible to define local variables using the `let` expression;
- Supports multiple dispatching, this means that in cases of overloaded operations, the decision of what operations is going to be called is based on the dynamic type of all parameters;
- Statically typed language;
- `Lists` and `Sets` are processed in a similar way to OCL expressions;
- The arithmetic expressions, Boolean expressions, and property access is similar to Java;
- It is possible to call Java operations;
- Simple types `String`, `Boolean`, `Integer`, and `Real` are similar to Java

The three languages use the same syntax, and all languages can operate on the same models, metamodels, and meta-metamodels. Therefore it makes the implementation using these three languages faster.

Only the Xtend language is used in this thesis for performing a M2M transformation. This language is commonly used with Xpand templates to provide reusable operations and simple expressions by extending the underlying metamodels [103]. A Xtend source code example is shown below. (Listing 2.14)

```
1  import myModel;
2  import Model;
3
4  Report root(GeneralReport aGeneralReport):
5
6    createReport(aGeneralReport);
7
8  create Report this createReport(GeneralReport aGeneralReport):
9    this.setName(aGeneralReport.Name)−>
10   this.setDate(aGeneralReport.Date);
```

Listing 2.14: Xtend example

In the example, the lines 1 and 2 import the two models used in the transformation. The `aGeneralReport` instance is passed as parameter in the transformation. This variable is an instance from the source model. The `createReport()` method, creates a new Report instance, based on the target model. The two attributes that are set in the new instance are the `Name` and `Date` attributes.

oAW also uses a workflow file, that is basically a Extensible Markup Language (XML) file, which is executed top down. The workflow guides the entire transformation, the metamodels and models, the M2M transformations calls using Xtend , the M2T transformations calls using Xpand, and the constraint checking calls using the Check language are defined

and configured in this file. The example below (Listing 2.15) shows the workflow used in Xtend implementation of the case study.

```
1  <workflow>
2    <property name="sourceModel" value="sourcemodel/MasterProject.uml" />
3    <property name="resultModel" value="result/result.xmi" />
4    <property name="resultDir" value="result" />
5    <property name="BSDMC_MM" value="metamodel/BSDMC_MM.ecore"/>
6    <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup" >
7      <platformUri value=".."/>
8      <registerEcoreFile value="metamodel/BSDMC_MM.ecore" />
9    </bean>
10     <bean class="org.eclipse.xtend.typesystem.uml2.Setup"
11         standardUML2Setup="true"/>
12     <component class="org.eclipse.xtend.typesystem.emf.XmiReader">
13         <modelFile value="${sourceModel}"/>
14         <outputSlot value="modelSlot"/>
15    </component>
16   <component class="org.eclipse.xtend.XtendComponent">
17     <metaModel class="org.eclipse.xtend.typesystem.uml2.UML2MetaModel"/>
18     <invoke value="template::UML2BSDMC_MM::root(modelSlot)"/>
19     <outputSlot value="outSlot"/>
20   </component>
21   <component id="resultModelWriter" class="org.eclipse.emf.mwe.utils.Writer">
22     <modelSlot value="outSlot"/>
23     <uri value="${resultModel}"/>
24   </component>
25 </workflow>
```

Listing 2.15: Workflow of the Xtend case study implementation

2.9.5. Object Constraint Language (OCL)

The declarative language OCL is used to define constraints and rules to models. It can be used to enhance the specification of models by setting constraints and conditions to specific elements. A big advantage of using OCL in MDE, is the possibility of validating and analyzing a model before generating its code. The OCL specification is defined by the OMG, the same group that defines specifications such as UML and QVT.

An example using OCL is by constraining the field `age` from an element from the type `Person` to a range from 0 to 120 years. It can also be used in custom templates to provide runtime behavior, initialize features in models, define model audits, and metrics, and serve as basis for transformation languages [103].

The OCL expressions can be included in the model or in the source code from one transformation language that supports OCL such as ATL, QVT, and EMF Model Query. In OCL, the `context` keyword introduces the context from one expression, so that using the keyword `self`, refers to the context of the expression. The keywords `inv`, `pre`, and `post` denotes the stereotypes invariant, precondition, and postcondition. It is also possible to specify the keyword `body` to specify a query. It may also be used to indicate an initial state

or a derived value from one attribute. The keywords used are `init` and `derive`. A detailed documentation about OCL can be found in [132]. A few examples of OCL expressions are shown below.

```
1 context Body
2  inv : name = 'EMB 190'
```

Listing 2.16: OCL example

In this first example, Listing 2.16, the context of the name attribute in a context `Body` must be ``‘‘EMB 190’’``. Another example of an OCL expression is shown in Listing 2.17.

```
1 context Source::Package::GetBehaviorDiagrams() : OrderedSet(Behavior)
2 body: allOwnedElements()−>select(e |
3         e.oclIsTypeOf(Activity) = true or
4         e.oclIsTypeOf(Interaction) = true)−>oclAsType(Behavior)−>asOrderedSet();
```

Listing 2.17: Another OCL example

The example above does not define a constraint, but it is a query that returns all `Activity`, and `Interaction` elements contained in the `allOwnedElements` set. This operation belongs to the context element that in this implementation is an element from the type `Package`. It returns an ordered set containing all filtered elements found.

3. Case Study

The objective of implementing this case study is to evaluate four transformation languages for UML quality engineering. This case study executes a M2M transformation, in which a set of software metrics are calculated in the source model, and based on this metrics a set of bad smells is searched for. The target model contains a list with all calculated metrics and bad smells detected in the source model. The figure below describes the case study process. (Figure 3.1)

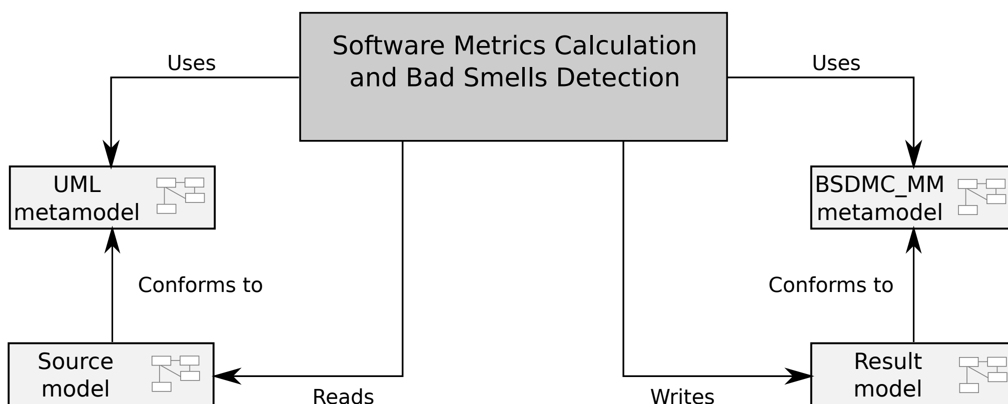

Figure 3.1.: Case study complete process

The transformation execution uses the source model, that is described in section 3.2, and the Bad Smell Detection and Metrics Calculation Metamodel (BSDMC_MM), described in section 3.1. When the transformation is executed, bad smells and the metric calculation results are saved in the *Result* model, which conforms to the BSDMC_MM metamodel.

This case study was implemented in four different transformation languages to analyze the languages features and characteristics. The implementations of the four different languages are described in detail in chapter 4.

The bad smells detected in the case study are *Method in Wrong Class*, *Field in Wrong Class*, and *Method Not in Use*, which are described in section 3.3. The metrics calculated are *Number of Children, Coupling Between Object Classes* , *Method Polymorphism*, *Method Related in Class*, and *Field Related in Class*, which are described in section 3.4.

3.1. Bad Smell Detection and Metrics Calculation Metamodel

The BSDMC_MM models the result structure that contains all necessary information from detecting, calculating and performing refactoring processes. Figure 3.2 shows the BS-DMC_MM specification.

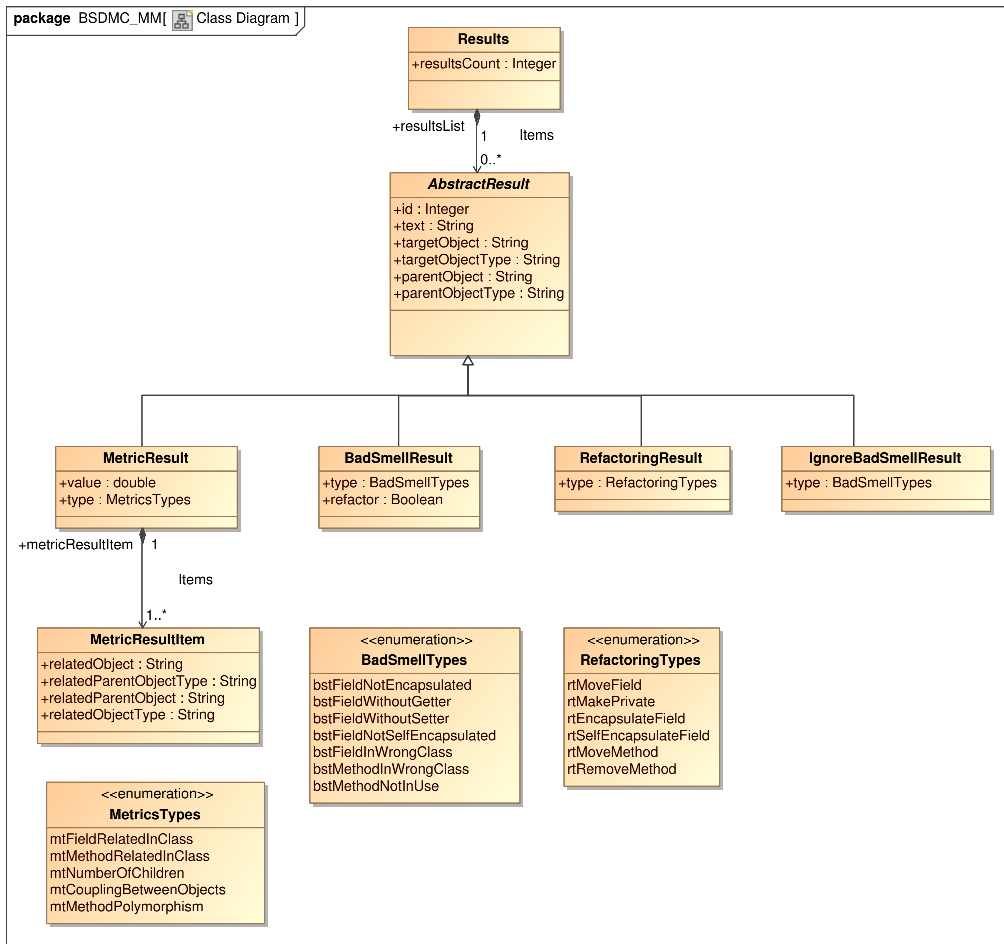

Figure 3.2.: BSDMC_MM Class Diagram

In the class diagram above, it is possible to observe that the metamodel structure contains a class named `Results`. This class contains one field that counts all result items found. It has a `0..*` cardinality to specialized objects from `AbstractResult`.
The abstract class `AbstractResult` contain all common fields for its specialized classes.

The field id is used to store a unique identifier for the result instance type. For example all metrics entries that calculate the metric *Number of Children* contains the id = 2. The text field is used to describe the entry purpose, or what the current entry means, as for example "Calculate Number of Children (NOC)". The other four fields, targetObject, targetObjectType, parentObject, and parentObjectType are used to identify the related element in the evaluated model. The specialized classes from AbstractResult are MetricResult, BadSmellResult, IgnoreBadSmellResult, and RefactoringResult.

The MetricResult specialization is used to store the information related to a metric calculation of a specific element. This class contains two fields, the value field used to store the metric value, and a type field that is an enumerated type from type MetricTypes that defines what metric is described by this instance. This specialization has a 1..* cardinality to the MetricResultItem class. This cardinality stores the information of what elements are related to the current element being analyzed by the metric. It contains the fields relatedObject, relatedObjectType, relatedParentObject, relatedParentObjectType, and they are used to identify the related element in the evaluated model.

The BadSmellResult class is instantiated if a bad smell has been detected, based on the MetricResult elements. It contains two fields, a type field, that is from enumerated type BadSmellTypes, and defines what bad smell this instance is describing. The refactor field, is a Boolean field, which as default is set as false. This field can be used in a refactoring phase to define what bad smells should be removed in the refactoring process.

Another specialized class is the IgnoreBadSmellResult, that is used to store what bad smells detection should be ignored for specific elements in the evaluated model. For example, do not create a bad smell entry *Method in Wrong Class* if the method that is been analyzed is *static*. These objects are instantiated in the metric calculation phase. It contains one enumerated type field named type. This field is from the BadSmellTypes, and it is used to define what bad smell should be ignored for one specific source model element. One advantage of using this class, is that in the bad smell detection phase, the source model is not used, but only the *Results* model. That, in cases of very large source models, can save time.

The RefactoringResult class is used to store the information about what refactoring has been performed in the model, based on the BadSmellResult elements that have the field refactor set to true.

Another metamodel for representing measurement information related to software, is the OMG Software Metrics Meta-Model (SMM) [129], and it is currently in its *Beta 1* version. SMM should provide an extendible metamodel, that can be used for exchanging information related to software measurements, concerning existing software assets such as designs, implementation, or operations. Compared to the BSDMC_MM metamodel used in the case study, the SMM specification is specially defined for software metrics, it can for example define scopes for metric calculations, metrics can be categorized in groups, where BSDMC_MM cannot. It provides much better support for metrics calculation than

BSDMC_MM, but it is also more complex. The BSDMC_MM metamodel was specified not only for metric calculations such as SMM, but also for bad smells detection and refactoring. In the BSDMC_MM specification there are elements specially defined for storing bad smells entries, and also refactoring process results. It has a much simpler specification than the SMM specification.

3.2. The Example Model

The example model used in the case study, models an airplane and its related objects. This model is used as the source model in the M2M transformation. The model contains one class diagram used to specify the structure of the airplane model described in section 3.2.1. Three *Activity Diagrams* are used to specify the behavior of specific methods in the model, and these are described in section 3.2.3. The *Sequence Diagram* is used to specify the behavior of the objects in its initialization. The *Sequence Diagram* is described in section 3.2.2.

3.2.1. The Class Diagram

The class diagram shown below (Figure 3.3), specifies the aircraft model structure. It consists of an abstract class `AbstractAircraft`, and three different specialized classes `FixedWingAircraft`, `ZeppelinAircraft`, and `RotorcraftAircraft`. The `X990` class models one airplane named "X990", and it is instantiated by the `Client` class. The `AbstractAircraft` class has one composition named `Engines` that defines that one aircraft may have an arbitrary amount of `engines`. It also has one `FuelTank` class, specified by the `FuelTank` composition, so that every aircraft must have a fuel tank.

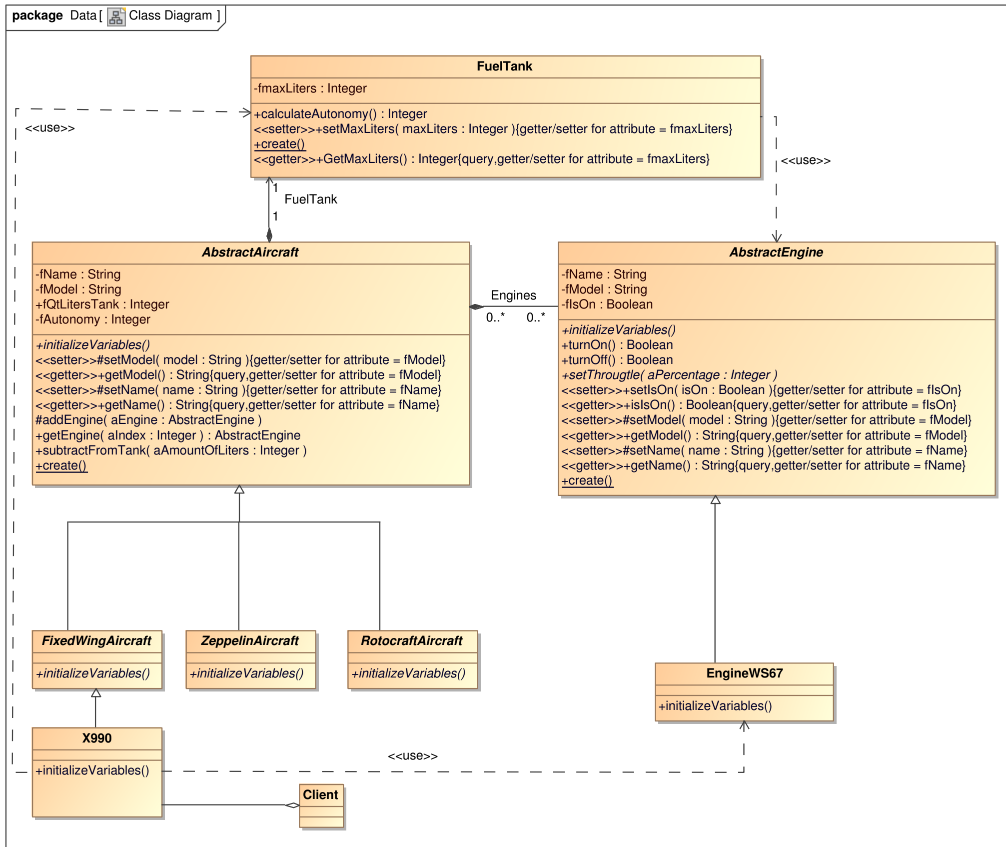

Figure 3.3.: Example model class diagram

3.2.2. The Sequence Diagram

This diagram describes the objects initialization. The class `Client` creates one instance of `X990`, and initialize its variables. The `X990` instance creates and initializes two instances of `EngineWS67` engines. After initializing the two engines, the `X990` instance initializes one instance of `FuelTank`. Figure 3.4 shows this initialization process.

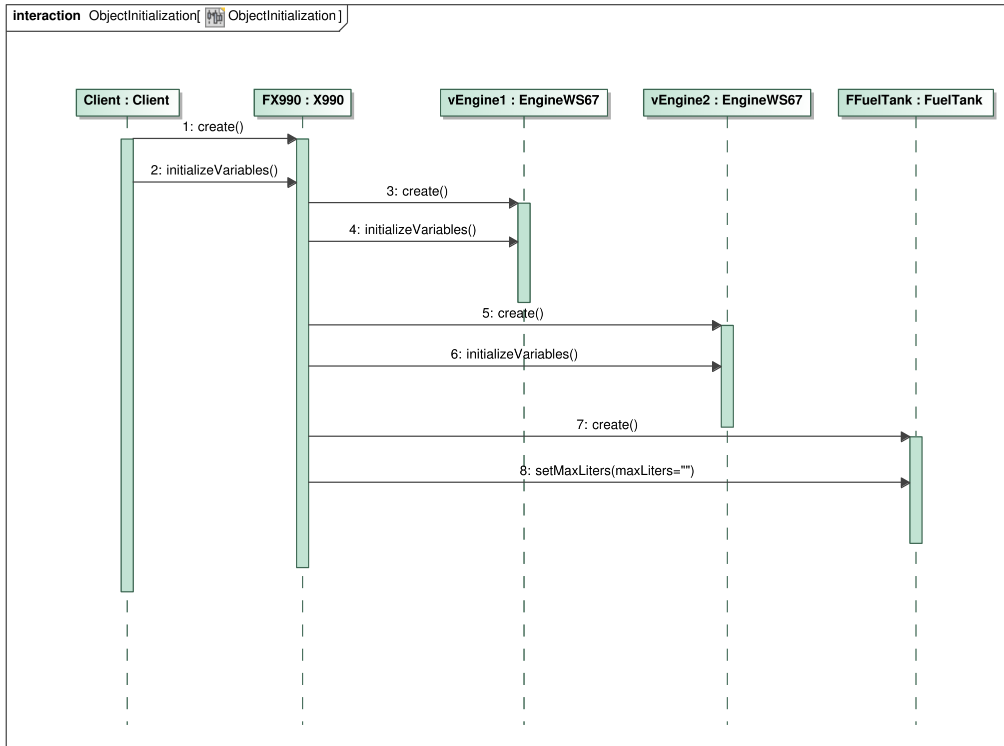

Figure 3.4.: Source model *Sequence Diagram* for initializing all objects

3.2.3. The Activity Diagrams

There are three different *Activity Diagrams*. Each *Activity Diagram* is connected to a method in the *Class Diagram*, so that they model the connected method execution behavior.

The *Activity Diagram* shown in Figure 3.5 models the `InitializeVariables` method from the `X990` class. The `InputPin` in the `CallOperationAction` `setModel`, `setName`, `addEngine`, `initializeVariables`, and `setMaxLiters` contains the parameter value from the method that is defined in the `Operation` field.
The `CreateObjectAction` elements, defines that an object from the type defined in the `Classifier` is going to be created, and the `Name` field is the variable name that will contain the object created in the function. According to [148], the `CreateObjectAction` is extended with stereotype `createObject` from the metaclass `CreateObjectAction`. These elements are identified in the diagrams by the `createObject` stereotypes.

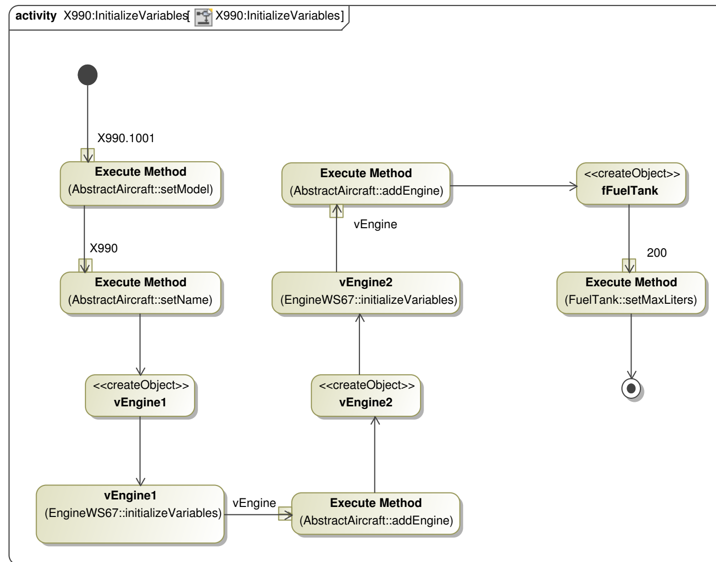

Figure 3.5.: *Activity Diagram* from method `X990::initializeVariables`

In the Figure 3.6 the *Activity Diagram* from the `CalculateAutonomy` method in the `FuelTank` class is shown. This diagram contains one `CreateObjectAction` that creates one instance of an `Integer` object for the `fmaxLiters` property. The `OpaqueAction` `Result =` calculates the return value from this method. The calculation formula is modeled in the `ValuePin`. In this object the formula is defined that is modeled in a `OpaqueExpression` object. It was used the `ValuePin` `name` field to map the property `fQtLitersTank` to its owner class. In this case, the field `fQtLitersTank` belongs to the `AbstractAircraft` class.

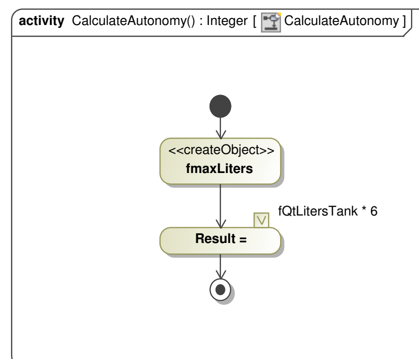

Figure 3.6.: *Activity Diagram* from method `FuelTank::CalculateAutonomy`

Another method that also contains one *Activity Diagram* is the `initializeVariables` method from the `EngineWS67` class. It is shown in the figure below (Figure 3.7). Only `CallOperationAction` actions are used in this diagram. In the initialization, the engine is turned off, throttle variable is set to zero, and the engine's name and model are set.

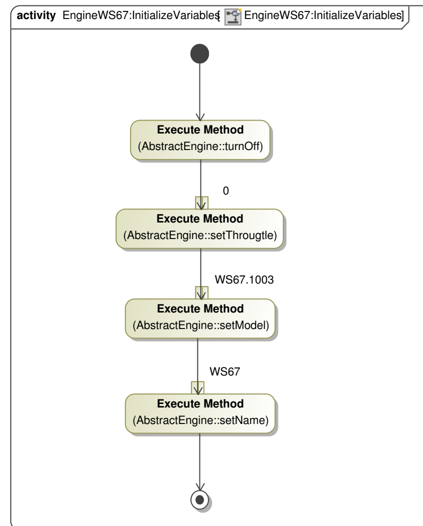

Figure 3.7.: *Activity Diagram* from method `EngineWS67::initializeVariables`

3.3. Bad Smell Detection

Three different bad smells types are detected in this implementation. The bad smells are calculated based only on the metrics calculated in the case study, and are described in section 3.4. The source model is not used in the bad smell detection phase. The bad smells that are detected in this implementation are *Method in Wrong Class*, *Field in Wrong Class*, and *Method Not in Use*. The *Method in Wrong Class* bad smell is detected in cases where the analyzed method is called more often in other classes than in the current class. The bad smell is ignored in cases where the method analyzed has a different behavior diagram related to it in its owner specialized classes (*Method Polymorphism* metric larger than 0), the method is a `getter` or `setter` from a property, or the method is *static*. The *Field in Wrong Class* bad smell is detected in cases that the analyzed field is called more often in other classes than in the current class. *Method Not in Use* bad smell is detected in cases that the method is not being used in any behavior diagram in the entire model. This bad smell is ignored in cases that the method is an *abstract* method.

3.4. Model Metrics

The metric calculation is executed before the bad smells are detected. It is executed first because the bad smell detection process is based only on the metric calculation results.

The two metrics *Number of Children* and *Coupling Between Object Classes* , that belong to the C.K. metrics are described in section 3.4.1 and section 3.4.2. The metric *Method Polymorphism* used for counting the method polymorphism is described in section 3.4.3. The metrics *Method Related in Class* and *Field Related in Class* used for calculating the number of times the method and the field are used, are described in section 3.4.4 and section 3.4.5.

3.4.1. *Number of Children*

The *Number of Children* metric belongs to the group of metrics known as C.K. metrics, and can be found in [85]. This metric is used to calculate how many specialized classes, the analyzed class has. Considering only the immediate subclasses of the analyzed class. According to [85], the greater the number of children, the greater is the reuse. But the likelihood of an improper abstraction of the parent class is also greater. The number of children also gives an idea of the influence of this class in the design.

In our case study, this metric is calculated by collecting all relationships from the type `Generalization` from the class that is being analyzed.
Taking the source model as example, the class `AbstractAircraft`, has as *Number of Children* metric value three, since the classes `FixedWingAircraft`, `ZeppelonAircraft`, and `RotorcraftAircraft` are the direct specializations from the `AbstractAircraft` class. The second level specialized class `X990` is not counted in this metric.

3.4.2. *Coupling Between Object Classes*

This metric also belongs to the C.K. metrics. According to [85], *Coupling Between Object Classes* relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. In our case study all relationships from one class to another are considered. So that we calculate the *Coupling Between Object Classes* based on the amount of relationships from one class to another, including generalizations.
One example is the coupling of the class `FixedWingAircraft`, that contains relationships to the classes `X990` and `AbstractAircraft`, so that the metric value is two.

3.4.3. *Method Polymorphism*

The *Method Polymorphism* metric is defined as the amount of rewriting of a specific method by its specialized classes. It is used in our case study as criteria to check if one specific

method is allowed to be moved to another class, in the case of bad smell detection. It is calculated by collecting all specialized classes, the entire tree down, and counting the number of times this method has a change in its behavior, by searching for behavior diagrams connected to the method analyzed in each of the methods owner specialized classes. The method polymorphism is characterized in our case study by a behavior diagrams attached to the method in a specialized class. One example is the `initializeVariables` method from the `abstractAircraft` class. The `X990` class, that is a specialized class from `abstractAircraft`, has a behavior diagram related to its `initializeVariables` method (Figure 3.5), so that the *Method Polymorphism* metric value is one.

3.4.4. *Method Related in Class*

Method Related in Class is a metric that is used to calculate the number of times the analyzed method is used by a different class, by navigating through the two types of behavior diagrams used in this case study, the *Activity* and *Sequence* diagrams.
The classes set that is going to be collected depends on the method analyzed visibility. If the method is *private*, only the owner class is analyzed. If the method is *protected*, then all specialized classes of its owner class are analyzed. If the method is *public*, then all classes that contain a relationship to the owner's method are collected, including the related classes of the owner specialized classes.
Using this set of classes, all *Activity Diagrams* used by methods in this classes set are navigating through to find if a behavior diagram is using the analyzed method. All model's *Sequence Diagrams* are also navigating through. The metric result is the number of times the method is being called in the two behavior diagrams types used in this case study. One example is the *Method Related in Class* metric from the method `setModel` from the class `AbstractEngine`. This method is a protected method from the `AbstractEngine` class. So that only the owner class itself and the specialized classes are analyzed. Observing the `AbstractEngine` does not call the method in any of its behavior diagrams, but the specialized class `EngineWS67` does once.

3.4.5. *Field Related in Class*

The *Field Related in Class* metric is used to calculate the number of times the analyzed field (`Property`) is called in each class related to its owner class. All *Activity Diagrams* used by the methods from the related classes are navigated through to the use of the analyzed field. One example is the *Field Related in Class* metric calculation from the `fmaxLiters` field. This field belongs to the `FuelTank` class, and it is only being used in the diagram `CalculateAutonomy` from its owner class.

4. Case Study Implementation

This chapter describes the case study implementation for each analyzed language. Section 4.1 gives an overview of the implementation structure and some implementation issues that are not language specific. In section 4.2 and section 4.3 the implementation of the two rule based transformation languages ATL and ETL are described. Section 4.4 describes the implementation for the imperative transformation language Operational QVT, while in section 4.5 is the implementation for the second imperative transformation language Xtend described.

4.1. Overview

All four implementations are similar in their function names, file names, including the implementation distribution through the files, and file locations. So that it is easier to understand one implementation once one of the other implementations is known. It is also possible and easier to create parallels between all four implementations. Language specific features were also implemented for each transformation language to better understand and evaluate its features.

The transformation files are described below:

- `UML2BSDMC_MM`: In this file the entry point for the transformation, and also the main methods and rules used in the transformation process are implemented.
- `metricCalc`: The implementation related to the creation of both elements, the BSDMC_MM::`MetricResult`, and the BSDMC_MM::`MetricResultItem` instances are implemented in this file.
- `ignoreBadSmells`: This file contains the implementation related to the creation of BSDMC_MM::`IgnoreBadSmellResult` instances.
- `badSmellDetect`: The implementation related to the creation of the element BSDMC_MM::`BadSmellResult` instances are implemented in this file.
- `functionsLib`: This file contains diverse functions that are used in the transformation.

Another important factor is the imperative implementation of the transformation languages Operational QVT and Xtend, compared to the hybrid approaches of ATL, and ETL. In the Operational QVT and Xtend implementation is the rule scheduling specified in the implementation, while in ATL and ETL transformation control flow defines what matched rules are called next. The three functions `CalculateMetricsInClasses`, `CalculateMetricsInOperations`, and `CalculateMetricsInProperties` were implemented in the hybrid approaches using *matched* rules, while the same functions in the imperative languages were called by our implementation. Another difference that exists between the other implementations and ATL is that it was necessary to implement two different transformation processes. It was tried to keep the two different implementations approaches as similar as possible.

The transformation result files are all saved in the XMI file format, and are located in the `result/` directory in each project. The following section describes how the metric *Field Related in Class* has been implemented for each of the transformation languages evaluated. The implementation relevant for the calculation of this metric is described in the sections below. The metric implementation has a medium complexity compared to the other four metrics implemented. All the bad smell detection implementations, all other metrics implementations, the creation of the ignore metrics rules implementation, build files, and languages special features implemented are all being omitted in this chapter.

4.2. ATLAS Transformation Language (ATL)

The ATL implementation is divided in two transformation processes. The first is responsible for the metric calculation and the instantiation of the ignore bad smell elements in the *Results* model, while the second transformation process is responsible for the bad smell detection. Between the four implementations, this is the only one that is divided in two transformation processes. The reason for dividing it into two different processes is that the matched transformation rules are called automatically by the transformation process, therefore it was not possible to force the execution of the bad smell detection after the metric calculation. One possible solution would be using the imperative block `do` from a matched rule that is executed when the element `source!Module` is found in the source model, and then implement all transformation in the `do` block. In this case *lazy* rules would be called in the imperative block, and the end result would be an imperative implementation similar to the Operational QVT, and Xtend implementations.

The Listing 4.1 shows the header from the file `UML2BSDMC_MM_p1.atl`, used in the first transformation process. The compiler `atl2006` used in this case study is defined in line 1. One reason for using this compiler is that for the language test reasons we implemented a rule inheritance, and it is only supported by this new compiler. In lines 2 and 3 the metamodels used in the transformation are set, followed by the module name declared

in line 4. In line 5 the transformation direction, target and source model declaration are defined. In line 7 the library `functionsLib.atl` used in the transformation is declared. Two *superimposed* modules `metricsCalc.asm`, and `ignoreBadSmells.asm` were included to this transformation, and these are the compiled files from the `metricsCalc.atl` and `ignoreBadSmells.atl`. *Superimposed* was used in this implementation to organize the rules better. In lines 9 and 10 the global variables that are used in the transformation are declared. The `fResult helper` is a reference to the target model, and the `fListBehaviorDiagrams helper` is a list containing *Activity*, and *Sequence* diagrams found in this case study. The `entrypoint` rule in line 12 is executed automatically by the transformation process before any other rule, so that this rule is used to initialize the `fResult` variable. The second transformation process is implemented in the `UML2BSDMC_MM_p2.atl` file, the difference is that the source model is not used in this transformation, since the bad smells are calculated based only on the *results* model.

```
1  -- @atlcompiler atl2006
2  -- @path QueryResult=/CaseStudy_ATL/metamodel/BSDMC_MM.ecore
3  -- @nsURI Source=http://www.eclipse.org/uml2/2.1.0/UML
4  module UML2BSDMC_MM_p1;
5  create OUT : QueryResult from IN : Source;
6
7  uses functionsLib;
8
9  helper def : fResult : QueryResult!Results = OclUndefined;
10 helper def : fListBehaviorDiagrams : OrderedSet(Source!Behavior) = OclUndefined;
11
12 entrypoint rule InitRule() {
13   to vqr: QueryResult!Results
14   do {
15     thisModule.fResult <- vqr;
16   }
17 }
```

Listing 4.1: ATL transformation initialization

The Listing 4.2 shows the implementation of the rule `CalculateMetricsInProperties`, that such as the rules `CalculateMetricsInClasses` and `CalculateMetricsInOperations`, is a matched rule that is called by the transformation process after the `entrypoint` rule was executed. In line 2 the guard condition to execute the rule is defined. This rule is only executed in this case study when the element analyzed is from type `Property`, and it does not have an aggregation and its name is different than an empty string, or it has a composite aggregation with an upper and lower bound equal to one. The property can have an aggregation, and its owner element must be from the type `Class`. So it is possible to filter diverse situations where the property should not be analyzed in our case study. The `using` block in line 8 is used to initialize three lists. One containing all classes related to the property owner (Line 9), one containing all specialized classes from the property owner class (Line 11), and the last list in line 12, implements the union of the two lists. In the imperative block `do` between the lines 15 and 26 are called the *lazy* rules for the *Field*

Related in Class metric calculation. In line 16, a feedback from the transformation in the console is given, so that the user can see the transformations order. In line 17, the *lazy* rule `CalculateMetricFRiC` is called that has as parameters the field that is being analyzed (variable `vs`), analyzed property owner class, and a list containing all behavior diagrams in the source model. In this line, the number of times the property is used in one diagram that belongs to its owner class is calculated. The result is included in target model in the list `thisModule.fResult.resultsList`. The same metric is calculated in line 21 for each of the classes related to the property's owner class. The result for each class is also included in the target models `resultsList` list.

```
1  rule CalculateMetricsInProperties {
2    from vs : Source!Property(vs.oclIsTypeOf(Source!Property) and
3                                     ((vs.aggregation = #none and vs.name <> '') or
4                                      ((vs.aggregation = #composite) and
5                                       (vs.lower = 1 and vs.upper = 1 ))) and
6                                    (vs.association = OclUndefined) and
7                                    (vs.owner.oclIsTypeOf(Source!Class)))
8     using {
9    vListRelatedClasses : OrderedSet(Source!Class) =
10                                      vs.owner.CollectRelatedClasses(vs.owner);
11   vListChildClasses : Set(Source!Class) = vs.owner.CollectAllChildClasses();
12   vListUnion : OrderedSet(Source!Class) =
13                                      vListRelatedClasses->union(vListChildClasses);
14    }
15   do {
16      vs.debug('Calculating metrics related to the property : ');
17      thisModule.fResult.resultsList <-
18        thisModule.CalculateMetricFRiC( vs,
19                                              vs.owner,
20                                              thisModule.fListBehaviorDiagrams);
21      thisModule.fResult.resultsList <- vListUnion->collect(e |
22        thisModule.CalculateMetricFRiC( vs,
23                                              e,
24                                              thisModule.fListBehaviorDiagrams));
25
26   }
27 }
```

Listing 4.2: ATL `CalculateMetricInProperties` rule

In Listing 4.3 the *lazy* rule that creates and initializes an instance of `QueryResult!MetricResult` is shown. This *lazy* rule is implemented in the `metricsCalc.atl` unit and is included in the transformation as a *superimposed* module. It is a *lazy* rule because it is called by the rule `CalculateMetricsInProperties` described above. In line 4 a new instance of a `QueryResult!MetricResult` is created. In the following lines, some of the properties from a new instanced element are set. In the `do` block, the *lazy* rule `CreateMetricItem` that creates an instance of a `QueryResult!MetricResultItem` element is defined. This element is initialized with the information related to the class passed as parameter. The information stored in this element is used to identify the related element to the metric calculation in the source model. This instance is included in the

vNewMetricResult.metricResultItem list. The actual metric calculation is performed in the helper CountRefBDFound in line 14. The result is the number of times the Property analyzed is referenced in the behavior diagrams that belongs to the class passed as parameter. The number of items in the thisModule.fResult.resultsList was increased by one in line 16. In line 17 the result element from the rule is referenced.

```
1  rule CalculateMetricFRiC(aProperty: Source!Property,
2                           aRelatedClass : Source!Class,
3                           aBehaviorDiagramsList : OrderedSet(Source!Behavior)) {
4    to vNewMetricResult:QueryResult!MetricResult(id <− 5,
5                                                 targetObject <− aProperty.name,
6                                                 targetObjectType <− Source!Property.name,
7                                                 parentObject <− aProperty.owner.getQualifiedName(),
8                                                 parentObjectType <− Source!Class.name,
9                                                 text <− 'Calculate Property Related in Class (PRiC)',
10                                                type <− #mtFieldRelatedInClass)
11   do {
12    vNewMetricResult.metricResultItem <−
13      thisModule.CreateMetricItem(aRelatedClass);
14    vNewMetricResult.value <− aProperty.CountRefBDFound(aRelatedClass,
15                                                        aBehaviorDiagramsList);
16    thisModule.fResult.resultsCount <− thisModule.fResult.resultsCount + 1;
17    vNewMetricResult;
18   }
19 }
```

Listing 4.3: ATL CalculateMetricFRiC rule

The fields are referenced in two different possible ways in our case study, it may be referenced in the body of an OpaqueExpression that belongs to one OpaqueAction, or it is referenced in the name field from a CreateObjectAction. This two element types, OpaqueExpression, and CreateObjectAction can only be found in *Activity Diagrams*. The helper in Listing 4.4 calls two other helpers, one for counting the number of times the property was found in OpaqueExpression, and other in CreateObjectAction elements. It is possible to observe that it was not necessary to pass the field as parameter, since the helper context is the field itself (Line 1). As parameter in both helper calls in lines 4 and 8 a filtered list containing only the behavior diagrams related to the aRelatedClass is passed as parameter in the CountRefDBFound helper in line 1. In both helper calls a filtered list is passed, containing the helper result is the sum of the result values of GetPropertyReferencesInOpaqueActions and GetPropertyReferencesInCreateObjectAction.

```
1  helper context Source!Property def :
2   CountRefBDFound(aRelatedClass : Source!Class ,
3                   aBehaviorDiagramsList : OrderedSet(Source!Behavior)): Real =
4    self.GetPropertyReferencesInOpaqueActions(
5                                             aBehaviorDiagramsList−>select(e |
6                                               e.oclIsTypeOf(Source!Activity))−>select(f |
7                                                 f.owner = aRelatedClass)) +
8    self.GetPropertyReferencesInCreateObjectAction(
```

```
9                                   aBehaviorDiagramsList−>select(e |
10                                      e.oclIsTypeOf(Source!Activity) and
11                                      e.owner = aRelatedClass));
```

Listing 4.4: ATL `CountRefDBFound` `helper`

The `helper` in Listing 4.5 counts the number of times the `context` element is used in the *Activity Diagrams* passed as parameter. The query starts on the diagrams list passed as parameter. In line 4 it is possible to observe that all nodes related to the diagrams are collected and then flattened. The `flatten()` function should always be called after collecting elements, so that the collected elements are flattened to be used in other selects. After collecting all nodes the `OpaqueAction` elements are filtered. All `input` elements from the `OpaqueAction` set (Line 5) are collected, flattened and then filtered again to create a new set, containing only elements in the `input` field from the `ValuePin` type, and that the property's `names` are equal to the `Property` analyzed owner name. This means that the `ValuePin` elements that are used in the behavior diagrams passed as parameter, in which the `ValuePin` `name` field is equal to the analyzed `Property` owner class name are collected. The elements in the `ValuePins` `value` property are collected and flattened.

This result set is filtered once again in which all `OpaqueExpression` elements that contain the `Property` analyzed name in the `name` field in its first body element. The resulting value is the remaining set count. It is possible to observe in line 6, that the `size()` value is an integer value, and must first be converted to string and then to real. ATL does not accept attributing an integer value to a real value. This `Query` was used to identify the use of the `fQtLitersTank` Property in the *Activity Diagram Calculate Autonomy* shown in Figure 3.6.

```
1 helper context Source!Property def :
2     GetPropertyReferencesInOpaqueActions(
3                                   aBehaviorDiagramsList : OrderedSet(Source!Behavior)) : Real =
4     aBehaviorDiagramsList−>collect(f | f.node)−>flatten()−>select(g |
5         g.oclIsTypeOf(Source!OpaqueAction))−>collect(h | h.input)−>
6             flatten()−>select(i | i.oclIsTypeOf(Source!ValuePin))−> select(j |
7                 j.name = self.owner.name)−>collect(l | l.value)−>flatten()−>select(m |
8                     m.oclIsTypeOf(Source!OpaqueExpression))−>select(n | n.body−>first()−>
9                         indexOf(self.name) <> −1)−>size()−>toString()−>toReal();
```

Listing 4.5: ATL `GetPropertyReferencesInOpaqueActions` `helper`

The `helper` shown in Listing 4.6 counts the number of times the `context` field is used in the *Activity Diagrams* passed as parameters. Such as in `helper` `GetPropertyReferencesInOpaqueActions`, the query is based on the diagrams passed as parameters. In this case all elements listed in each `node` attribute from each behavior diagram is collected. This set is flattened and then all elements of the type `CreateObjectAction` are selected in line 5. A new select is now used to filter all `CreateObjectAction` that have the same name as the element that is being analyzed name, and the classifier corresponds to the analyzed fields owner. This `Query` was used to identify the use of the `fmaxLiters` Property in the *Activity Diagram Calculate Autonomy* shown in Figure 3.6.

```
1 helper context Source!Property def :
2    GetPropertyReferencesInCreateObjectAction(
3                                          aBehaviorDiagramsList : OrderedSet(Source!Behavior)) : Real =
4    aBehaviorDiagramsList−>collect(f | f.node)−>flatten()−>select(g |
5       g.oclIsTypeOf(Source!CreateObjectAction))−>select(h |
6          h.name = self.name and h.classifier = self.owner)−>size()−>toString()−>toReal();
```

Listing 4.6: ATL `GetPropertyReferencesInCreateObjectAction` helper

4.3. Epsilon Transformation Language (ETL)

The ETL transformation process is composed only of one transformation. Similar to ATL, ETL has a block that is executed before all other matched rules are called, but in ETL it is used the block `pre`, and not a matched rule. Differently from ATL, is the block `post` that is executed after all matched rules have been called. For that reason it was not necessary to implement a second transformation process as in ATL. The bad smell detection is implemented in the `post` block in the ETL implementation. The main transformation file is the `UML2BSDMC_MM.etl`, and its header and relevant implementation for the *Field Related in Class* metric calculation is shown in Listing 4.7. The transformation is started by an Apache Ant [3] file, similar to Xtend workflows. From line 1 to 4 all libraries that are used in the transformation are declared. Between lines 6 and 10 the `pre` block where all global variables used are defined, is implemented. This block is automatically executed before all other matched rules are called. Line 7 is used to give a feedback from the transformation in the console, informing that the transformation processes has started. In lines 8 and 9 the global variables used in this transformation are declared. The first matched rule that is called is the `CreateDocumentRoot` declared in line 12. This rule initializes the two global variables. It was possible to observe by testing that the transformation process executes this rule before the others, probably because the `Model` element is closer to the model root element in the source model.

```
1  import 'functionsLib.etl';
2  import 'ignoreBadSmells.etl';
3  import 'badSmellDetect.etl';
4  import 'metricCalc.etl';
5
6  pre {
7     'Transformation started...'.println();
8     var fResult : QueryResult!Results = null;
9     var fListBehaviorDiagrams : OrderedSet(Source!Behavior) = null;
10 }
11
12 rule CreateDocumentRoot
13    transform vs : Source!Model
14    to vModel : QueryResult!Results{
15       fResult := vModel;
16       fListBehaviorDiagrams := vs.GetBehaviorDiagrams();
17    }
```

Listing 4.7: ETL transformation initialization

Listing 4.8 shows the implementation of the matched rule `CalculateMetricsInProperties`. Such as the rules `CalculateMetricsInClasses`, and `CalculateMetricsInOperations`, they are called automatically by the transformation process. It is possible to observe similarities between ETL and ATL. The `transform` block in line 2 defines the matching condition for this rule to be executed. The `to` block works very similar to the ATL implementation, in this case one instance of a `QueryResult!MetricResult` element is created and assigned to the variable `vNewMetricResult`. The difference to ATL here is that the filtering condition to execute the rule is defined in a guard block. The same guard condition as in ATL also applies. In line 10 again one output line in the console is generated, informing the user which property is being analyzed. The implementation from lines 11 to 16 are similar to the implementation in ATL for creating the lists containing the related classes. In line 18 the `Operation` `CalculateMetricFRiC` is called for calculating the *Field Related in Class* metric from the property analyzed, related to the class passed as parameter in the `Operation`, that in this case is the analyzed `Property` owner class. The list containing all behavior diagrams is also passed as parameter. The `Operation` result is added to the target model `fResult.resultsList`. For each related class in the `vListUnion` list, one new instance of `QueryResult!MetricResult` is created, and the metric *Field Related in Class* related to the lists current class is calculated.

```
1  rule CalculateMetricsInProperties
2      transform vs : Source!Property
3      to vNewMetricResult : QueryResult!MetricResult {
4      guard : ((vs.aggregation = Source!AggregationKind#none and
5                vs.name <> '') or
6               ((vs.aggregation = Source!AggregationKind#composite) and
7                (vs.lower = 1 and vs.upper = 1)) and
8               (vs.association.isUndefined()) and
9               (vs.owner.isTypeOf(Source!Class)))
10     null.println('Calculating metrics related to the property : ' + vs.name);
11     var vListRelatedClasses : OrderedSet(Source!Class) =
12       vs.owner.CollectRelatedClasses(vs.owner);
13     var vListChildClasses : Set(Source!Class) =
14       vs.owner.CollectAllChildClasses();
15     var vListUnion : OrderedSet(Source!Class);
16     vListUnion := vListRelatedClasses->includingAll(vListChildClasses);
17     fResult.resultsList.add(
18       CalculateMetricFRiC(vNewMetricResult,
19                           vs,
20                           vs.owner,
21                           fListBehaviorDiagrams));
22     fResult.resultsList.addAll(vListUnion->collect(e |
23       CalculateMetricFRiC(new QueryResult!MetricResult,
24                           vs,
25                           e,
26                           fListBehaviorDiagrams)));
27 }
```

Listing 4.8: ETL `CalculateMetricInProperties` rule

Different from the ATL implementation, ETL uses *Operation* that can be used as a function, accepting a sequence of instructions. In Listing 4.9 it is possible to observe that an instance of `QueryResult!MetricResult` is passed as parameter in line 1, as also the property that is being analyzed in line 2. The other two parameters are the related class that is the class in which the `Operation` will search for references to the `Property` analyzed, and a list containing the source model behavior diagrams. A different way of implementing this `operation` would be using as context the `Property` itself, and referring to the analyzed `Property` as `self`, and not as `aProperty`. In line 14 such as in line 12 in Listing 4.3, one instance of `QueryResult!MetricResultItem` is created. The metric value is calculated by an `Operation` named `CountRefBDFound`. The target model variable that counts the number of items in `resultsList` is increased by one in line 17. In line 18 the element that is returned in this `Operation` is declared.

```
1  operation CalculateMetricFRiC(aMetricResult : QueryResult!MetricResult,
2                                aProperty : Source!Property,
3                                aRelatedClass : Source!Class,
4                                aBehaviorDiagramsList : OrderedSet(Source!Behavior)) :
5                                    QueryResult!MetricResult{
6    aMetricResult.id := 5;
7    aMetricResult.targetObject := aProperty.name;
8    aMetricResult.targetObjectType := Source!Property.name;
9    aMetricResult.parentObject := aProperty.owner.getQualifiedName();
10   aMetricResult.parentObjectType := Source!Class.name;
11   aMetricResult.text := 'Calculate Property Related in Class (PRiC)';
12   aMetricResult.type :=
13           BSDMC_MM!MetricsTypes#mtFieldRelatedInClass.instance;
14   aMetricResult.metricResultItem.add(CreateMetricItem(aRelatedClass));
15   aMetricResult.value := aProperty.CountRefBDFound(aRelatedClass,
16                                                    aBehaviorDiagramsList);
17   fResult.resultsCount := fResult.resultsCount + 1;
18   return aMetricResult;
19 }
```

Listing 4.9: ETL `CalculateMetricFRiC operation`

The `operation` in Listing 4.10, implements the same routine as the `helper CountRefBDFound` in ATL Listing 4.4. The only difference is in the syntax, but the behavior is exactly the same.

```
1  operation Source!Property CountRefBDFound(aRelatedClass : Source!Class,
2                                            aBehaviorDiagramsList : OrderedSet(Source!Behavior)): Real {
3   return self.GetPropertyReferencesInOpaqueActions(
4                                               aBehaviorDiagramsList−>select(e |
5                                                   e.isTypeOf(Source!Activity))−>select(f |
6                                                       f.owner = aRelatedClass)) +
7          self.GetPropertyReferencesInCreateObjectAction(
8                                               aBehaviorDiagramsList−>select(e |
9                                                   e.isTypeOf(Source!Activity) and
10                                                  e.owner = aRelatedClass)).asReal();
11 }
```

Listing 4.10: ETL `CountRefDBFound operation`

The two Listings 4.11 and 4.12 are very similar to the ATL implementation in Listings 4.5 and 4.6. It is again only a syntax issue, since the queries are implemented exactly the same way. The only relevant difference is in line 8 from Listing 4.5. To find a *substring* in a *string* in ETL, the function `isSubstringOf()` is used, while in ATL the function `indexOf()` is used. It was not possible to find a similar function in ATL. Another difference is to typecast one variable or set, in ETL the `isTypeOf` function is used, different from all other implementations analyzed.

```
1  operation Source!Property GetPropertyReferencesInOpaqueActions(
2                                aBehaviorDiagramsList : OrderedSet(Source!Behavior)) : Real {
3   return aBehaviorDiagramsList−>collect(f | f.node)−>flatten()−>select(g |
4              g.isTypeOf(Source!OpaqueAction))−>collect(h | h.input)−>
5                  flatten()−>select(i | i.isTypeOf(Source!ValuePin))−> select(j |
6                      j.name = self.owner.name)−>collect(l | l.value)−>flatten()−>select(m |
7                          m.isTypeOf(Source!OpaqueExpression))−>select(n |
8                              self.name.isSubstringOf(n.body−>first().toString()))−>size()
9                                  −>toString()−>asReal();
10 }
```

Listing 4.11: ETL `GetPropertyReferencesInOpaqueActions` operation

```
1  operation Source!Property GetPropertyReferencesInCreateObjectAction(
2                                aBehaviorDiagramsList : OrderedSet(Source!Behavior)) : Real {
3   return aBehaviorDiagramsList−>collect(f | f.node)−>flatten()−>select(g |
4            g.isTypeOf(Source!CreateObjectAction))−>select(h |
5                h.name = self.name and h.classifier = self.owner)−>size()−>toString()−>asReal();
6  }
```

Listing 4.12: ETL `GetPropertyReferencesInCreateObjectAction` operation

4.4. Operational QVT Language

The complete implementation in QVT was done using the imperative language OML, more specifically the Eclipse Operational QVT, available in the Eclipse M2M project. All references in this thesis to Operational QVT refer to the Eclipse implementation of Operational QVT [28]. This implementation is different from the two implementations above described by the fact that ATL and ETL languages are hybrid languages, supporting imperative and declarative blocks. Only the OML language was used in this implementation, therefore it is an imperative implementation, also known as *operational transformation*. A possible way of archiving an implementation that is similar to ATL and ETL would be a QVT hybrid implementation, using also the declarative languages from QVT *relation language* and/or *core language*. This master thesis only analyzes the imperative language of QVT.

There is also only one transformation process, and it is implemented in the `UML2BSDMC_MM.qvto` file, that is partially shown in Listing 4.13. All libraries used are declared between lines 1 and 4. In lines 6 and 7, the used metamodels are declared. The reference to the two models is defined in the Eclipse IDE. In line 9 the transformation direction,

the models used, and the metamodels are declared. The transformation entry point is the `main` function in line 11. In line 12 is specified that for each `Model` element found in the `rootObjects` list from the `IN Source` model, the `mapping CreateDocumentRoot` is called. The `CreateDocumentRoot mapping` declared in line 15 has as context the `Model` element, that means that referring to `self` in the `mapping` is the same as the model mapped in line 12. This function result is an instance of `BSDMC_MM::Results`, that is the entire target model.

The implementation in line 17 writes a string, the context element `self`, and the log level in the console are passed as parameter in the `log` function, so that the user has a feedback from the transformation in the console. In line 18 all the sub elements from the model are filtered by the type `Class`, so that the result class is a set containing all `Class` elements found in the `Model` element. The `xcollect` function calls the `mapping CalculateMetricsInClasses` for each class contained in the set. As parameters the `result` variable, that is a reference from the target model, and the class that is going to be analyzed are passed. One interesting characteristic is the typecast. For example, the `e` variable must be typecast so that the transformation processes knows what type of elements it is. In this line we used `e.oclAsType(Class)`. Another way of typecasting was implemented in line 20. In this case the typecast was done in the result set from the `allSubobjectsOfType` function with `[Class]`. If the `e` variable is observed in line 21, it is possible to observe that a typecast was not needed in this case. It is in our point of view a strange behavior, since by filtering the set with the function `allSubobjectsOfType`, it is obvious that the result set is a set of the element type passed as parameter. Between lines 20 and 25 is implemented the routine that filters all `Operation` element types from the model, and calls the `CalculateMetricsInOperations` function for each element from the type `Operation` found. The implementation that calls the `mapping CalculateMetricsInProperties` is implemented between lines 27 and 35.

It is possible to observe between lines 29 and 32 the filtering condition for calculating the metric *Field Related in Class*. It is exactly the same logic as it was implemented in the `guard` condition in ETL, or in the `from` condition in ATL. Another possible way of implementing this in our case study, is to write this logic in a `when` block in the `mapping CalculateMetricFRiCs` in Listing 4.15. Each element from type `Property` found in each class that satisfies the select between lines 28 and 32 is passed as parameter to the `mapping CalculateMetricsInProperties`. As parameters the `result` variable, that contains the reference to the `BSDMC_MM::Results` element from the target model, the `Property` that is analyzed, and the list containing *Activity* and *Sequence* diagrams used in this case study are passed. In line 38 is set the property `resultsCount` from the target model, containing the amount of items found in the list containing all metrics calculated, bad smells detected, and ignore bad smell entries. The function from line 39 is exactly the same as in line 17, the difference is the `when` keyword after the `log` function is used to define a condition, in what circumstances should the log text be printed out. It is also possible to use assert such as it is shown in lines 41 and 42. The Operational QVT from Eclipse does not provide a

debugger. A workaround to it, is to use the `log` instructions with conditions, and `asserts`.

```
1  import metricsCalc;
2  import badSmellDetect;
3  import ignoreBadSmells;
4  import functionsLib;
5
6  modeltype BSDMC_MM uses 'http://BSDMC_MM.ecore';
7  modeltype Source uses 'http://www.eclipse.org/uml2/2.1.0/UML';
8
9  transformation UML2BSDMC_MM(in IN:Source, out BSDMC_MM);
10
11 main() {
12    IN.rootObjects()[Model] -> map CreateDocumentRoot();
13 }
14
15 mapping Source::Model::CreateDocumentRoot(): BSDMC_MM::Results {
16
17       log('Transformation started...',self,1);
18       self.allSubobjectsOfType(Class)->xcollect(e |
19          map CalculateMetricsInClasses(result, e.oclAsType(Class)));
20       self.allSubobjectsOfType(Class)[Class]->xcollect(e |
21          e.getAllOperations()[Operation]->select(f |
22             f.owner = e)->xcollect(g |
23                map CalculateMetricsInOperations(result,
24                                                  g,
25                                                  self.GetBehaviorDiagrams())));
26
27       self.allSubobjectsOfType(Class)[Class]->xcollect(e |
28             e.getAllAttributes()[Property]->select(f |
29                ((f.aggregation = uml::AggregationKind::none and f.name <> '') or
30                 ((f.aggregation = uml::AggregationKind::composite) and
31                   (f.lower = 1 and f.upper = 1 ))) and
32                (f.owner = e) and (f.association = null))->xcollect(g |
33                   map CalculateMetricsInProperties(result,
34                                                    g,
35                                                    self.GetBehaviorDiagrams())));
36       // THE BAD SMELL DETECTION IMPLEMENTATION WAS REMOVED IN THE MASTER THESIS
37       // TO SAVE PLACE.
38       resultsCount := result.resultsList->size();
39       log('Transformation finished succesfully...') when resultsCount = 292;
40       assert error (resultsCount = 292) with
41          log('Error in the transformation execution!',self,3);
42 }
```

Listing 4.13: Operational QVT transformation initialization

The `mapping` described in Listing 4.14 is used to call the `mappings` that calculate the metric *Field Related in Class* by calling the `mapping CalculateMetricFRiCs` in line 5, and for calling the `mapping` responsible for creating a `BSDMC_MM::IgnoreBadSmellsResults` instance if necessary. In line 1 it is possible to observe the use of `inout` from the parameter `aResults` that is from type `BSDMC_MM::Results`. In this case, the parameter `aResults` passed as parameter can be read and written as a parameter passed by reference. In line 5 the property that is analyzed maps the `mapping CalculateMetricFRiCs`, this property is the context of

the `mapping` described in Listing 4.15.

```
1 mapping CalculateMetricsInProperties(inout aResults : Results,
2                                     in aProperty :Property,
3                                     in aBehaviorDiagramsList : OrderedSet(Behavior)){
4   log('Calculating metrics related to the property : ' + aProperty.name);
5   aProperty.map CalculateMetricFRiCs(aResults,aBehaviorDiagramsList);
6   aProperty.map IgnoreBadSmellInProperty(aResults);
7 }
```

Listing 4.14: Operational QVT `CalculateMetricInProperties` `mapping`

The source code in Listing 4.15, shows the implementation of two `mappings`. It is possible to observe in the first `mapping` the use of the `init` block, that is used to initialize two lists, one containing all specialized classes from the `Property` analyzed owner class, and the other containing all classes related to the `Property` owned class. In line 11 the call of the `mapping` `CalculateMetricFRiC` is implemented. It is responsible for creating one instance of a `BSDMC_MM::MetricResult`. The result element is added to the `aResults.resultList` list. As parameters the property owner class, and the behavior diagrams list are passed. In line 14 are the two initialized lists in the `init` block concatenated, and for each class found in the concatenated list, the `mapping` `CalculateMetricFRiC` is called once. Each new `BSDMC_MM::MetricResult` is also included to the `aResults.resultList` list. In line 19 the `CalculateMetricFRiC` `mapping` is declared. Between lines 22 and 32 the return element properties are set. In line 29 the `query` that calculates the number of times the field is used in the behavior diagrams from the class passed as parameter is called. The implementation is similar to the ATL, and ETL implementation, and it is described in Listing 4.16. In line 32 one instance of `BSDMC_MM::MetricResultItem` containing the information of the related class is created.

```
1  mapping Source::Property::CalculateMetricFRiCs(
2                        inout aResults : Results,
3                        in aBehaviorDiagramsList : OrderedSet(Behavior)) {
4   init{
5     var vListChildClasses : Set(uml::Class) :=
6            self.owner.oclAsType(Class).CollectAllChildClasses();
7     var vListRelatedClasses : Set(uml::Class) :=
8            self.owner.oclAsType(Class).CollectRelatedClasses(
9                                     self.owner.oclAsType(Class));
10  }
11  aResults.resultsList += self.map CalculateMetricFRiC(
12                        self.owner.oclAsType(Class),
13                        aBehaviorDiagramsList);
14  vListChildClasses−>union(vListRelatedClasses)−>xcollect(e |
15    aResults.resultsList += self.map CalculateMetricFRiC(e.oclAsType(Class),
16                          aBehaviorDiagramsList));
17 }
18
19 mapping Source::Property::CalculateMetricFRiC(
20                  in aRelatedClass : Class,
21                  in aBehaviorDiagramsList : OrderedSet(Behavior)) : BSDMC_MM::MetricResult{
22   id := 5;
```

```
23   text := 'Calculate Field Related in Class (FRiC)';
24   targetObject := self.name;
25   targetObjectType := self.metaClassName();
26   parentObject := self.owner.oclAsType(Class).getQualifiedName();
27   parentObjectType := self.owner.oclAsType(Class).metaClassName();
28   type := MetricsTypes::mtFieldRelatedInClass;
29   value := CountRefBDFound(self,
30                            aRelatedClass,
31                            aBehaviorDiagramsList);
32   result.metricResultItem += result.map CreateMetricItem(aRelatedClass);
33 }
```

Listing 4.15: Operational QVT `CalculateMetricFRiCs` and `CalculateMetricFRiC` `mappings`

The `query` in Listing 4.16, implements the same routine as the `helper CountRefBDFound` in ATL Listing 4.4, and the ETL `Operation` in Listing 4.10. In the QVT implementation, differently from the other two, the implemented `query` does not have a context, so that the field was passed as parameter (`aProperty`). Other obvious difference is the syntax between the three implementations.

```
1 query CountRefBDFound(in aProperty : Property,
2                       in aRelatedClass : Class ,
3                       in aBehaviorDiagramsList : OrderedSet(Behavior)): Real {
4   return aProperty.GetPropertyReferencesInOpaqueActions(
5             aBehaviorDiagramsList[Behavior]−>select(e | e.owner = aRelatedClass)) +
6          aProperty.GetPropertyReferencesInCreateObjectAction(
7             aBehaviorDiagramsList[Behavior]−>select(e | e.owner = aRelatedClass));
8 }
```

Listing 4.16: Operational QVT `CountRefDBFound` `query`

Listings 4.17 and 4.18 implement exactly the same logic as the routines from the two already described languages. Taking as example the Listings below, it is possible to observe how compact the query is in relation to the other three implementations analyzed. Analyzing the filtering of *Activity Diagrams* in line 3, it is possible to observe that the selection of all *Activity Diagrams* in the list is done by the *[Activity]* construct, in which *Activity* is the element type being filtered in the *aBehaviorDiagramsList* list. In all other implementations it is necessary to build a complete select to filter the *Activity* elements from the list. Three further differences found in relation to the other implementations, is how the `Body` element from the `OpaqueExpression` is accessed (Line 6). In all other implementations, the element accessed is the `Body` element, but in QVT the element `_body` must be accessed. Finding the substring in the `name` field is also different than in all other implementations, as is shown in line 6. The typecasting function `oclIsTypeOf` is equal to ATL, but not to the other two languages.

```
1 query Source::Property::GetPropertyReferencesInOpaqueActions(
2                          in aBehaviorDiagramsList : OrderedSet(Behavior)) : Real {
3   return aBehaviorDiagramsList[Activity].node[uml::OpaqueAction].
4             input[uml::ValuePin]−>select(e | e.name = self.owner.oclAsType(Class).name).
```

```
5         value[uml::OpaqueExpression]−>select(f |
6             f._body−>first().find(self.name) > 0)−>size();
7 }
```

Listing 4.17: Operational QVT `GetPropertyReferencesInOpaqueActions` query

```
1 query Source::Property::GetPropertyReferencesInCreateObjectAction(
2                     in aBehaviorDiagramsList : OrderedSet(Behavior)) : Real {
3   return aBehaviorDiagramsList[Activity].node[uml::CreateObjectAction]−>select(e |
4         e.name = self.name and e.classifier = self.owner)−>size();
5 }
```

Listing 4.18: Operational QVT `GetPropertyReferencesInCreateObjectAction` query

4.5. Xtend

The transformation process is called by a oAW workflow file. In the workflow file all models and metamodels used, the transformation entry point file `UML2BSDMC_MM.ext`, and the entry point function `root` are declared. Xtend is an imperative language, so it does not provide the declarative features that ATL and ETL do. There are similarities in the implementation between QVT and Xtend, since both are imperative languages. The Listing 4.19 is a partial version of the `UML2BSDMC_MM.ext` file listed. In lines 1 and 2 the two metamodels used in the transformation process are referenced. From lines 3 to 6 the extension files are imported. In line 8 the entry point function `root` defined in the workflow is declared, this function returns one instance of `BSDMC_MM::Results` type, that is the entire target model result. The function `createDocumentRoot` that is called in line 9 is the main function in this implementation. In this function the calls for calculating all metrics and bad smells detections are implemented. The `create` keyword in line 11 defines that one instance of a `BSDMC_MM::Results` element is created and returned at the end of the function execution. As parameter the variable `aData` that contains the source model's element from type `Model` is also passed. In line 12 the function `GetBehaviorDiagrams` that collects all behavior diagrams from types *Activity* and *Interaction* is called. The returned set is stored in the variable `vBehaviorDiagramsList`. Such as in the other three implementations, it is also possible to print text in the console using `println`. Such as in the QVT implementation, from lines 14 to 22 the functions for calculating the metrics related to classes and to operations are called. In line 18 the `and` clause, expressed as && in Xtend is declared. The `CalculateMetricsInOperations` function, is only called if the first expression of the `and` clause is satisfied. The implementation from lines 23 to 33 are also very similar to the QVT implementation. The properties filtering from lines 25 to 29 have exactly the same logic as in all other implementations. The `BSDMC_MM::Results.resultsList` size is set in the `BSDMC_MM::Results.ResultsCount` field in line 37.

```
1  import uml;
2  import BSDMC_MM;
3  extension template::metricsCalc;
4  extension template::functionsLib;
5  extension template::badSmellDetec;
6  extension template::ignoreBadSmell;
7
8  Results root(Model aData):
9     createDocumentRoot(aData);
10
11 create Results this createDocumentRoot(Model aData):
12    let vBehaviorDiagramsList = GetBehaviorDiagrams(aData):
13    println('Transformation started...')−>
14    aData.packagedElement.typeSelect(Class).select(c |
15        CalculateMetricsInClasses(c,this))−>
16    aData.packagedElement.typeSelect(Class).select(c |
17         c.getAllOperations().typeSelect(Operation).select(e |
18            (e.owner == c) &&
19            CalculateMetricsInOperations(c,
20                                         e,
21                                         this,
22                                         vBehaviorDiagramsList) == true))−>
23    aData.packagedElement.typeSelect(Class).select(c |
24        c.getAllAttributes().typeSelect(Property).select(e |
25            ((e.aggregation == uml::AggregationKind::none) &&
26              (e.name != '') | |
27              ((e.aggregation == uml::AggregationKind::composite) &&
28               (e.lower == 1 && e.upper == 1))) &&
29             (e.owner == c) && (e.association == null) &&
30             CalculateMetricsInProperties(c,
31                                          e,
32                                          this,
33                                          vBehaviorDiagramsList) == true))−>
34    // THE BAD SMELL DETECTION IMPLEMENTATION WAS REMOVED IN THE MASTER THESIS
35    // TO SAVE PLACE.
36    println('Transformation finished succesfully...')−>
37    this.setResultsCount(this.resultsList.size);
```

Listing 4.19: Xtend transformation initialization

In Listing 4.20, the implementation of the `CalculateMetricsInProperties` function is shown. As parameter the properties owner class, the property itself, the target model, and the list containing all behavior diagrams are passed. Line 6 shows the called function `CalculateMetricFRiC` that calculates the metric *Field Related in Class* and creates instances of a `BSDMC_MM::MetricResult` containing the metric result calculation. In line 8 the function that checks, and if necessary, creates an instance of `BSDMC_MM::IgnoreBadSmellsResults` is called.

```
1  CalculateMetricsInProperties(Class aClass,
2                               Property aProperty,
3                               Results aResults,
4                               List aBehaviorDiagramsList):
5     println('Calculating metrics related to the property : ' + aProperty.name)−>
6       CalculateMetricFRiCs(aProperty,aClass,aResults,aBehaviorDiagramsList)−>
```

```
7      println('Detecting Ignore Bad Smell conditions for property : ' + aProperty.name)−>
8      IgnoreBadSmellsInProperty(aClass,aProperty,aResults,aBehaviorDiagramsList);
```

Listing 4.20: Xtend `CalculateMetricInProperties` function

Listing 4.21 shows the implementation of the function `CalculateMetricFRiCs` and `CalculateMetricFRiC`. In line 5 one `List` element that is passed as parameter in the function `CollectRelatedClasses` is created. In line 6 a variable is created that contains a list containing all specialized classes from the properties owner class passed as parameter. In line 9 is created a variable that contains all related classes to the properties owner classed passed as parameter. We detected that when a function is called more then once with the same parameters values, the function is not executed again. For this reason, one extra parameter was included in the functions `CollectAllChildClasses`, and `CollectRelatedClasses` to guarantee that each function is never called with the same parameters twice. Another difference is in the function `CollectRelatedClasses`, where two extra parameters are passed. The result set is exactly the same as in the other implementations, but the function was implemented differently. In line 13 the function `CalculateMetricFRiC` that calculates the *Field Related in Class* metric from the property that is being analyzed is called. This function is implemented between lines 36 and 50 and it returns an instance of `BSDMC_MM::MetricResult` containing the metric calculation results. These instance is added to the list `BSDMC_MM::Results.resultsList` from the target model. In this function it is again possible to observe that it has one extra parameter to guarantee the functions execution.

From line 18 to 23 a select clause is implemented, in which for each specialized class found, the function `CalculateMetricFRiC` is called once. Each created instance is included in the `BSDMC_MM::Results.resultsList` list. The same occurs in the implementation from line 24 to line 31. But in this case, only the related classes are included, and not the specialized classes. The implementation between those lines could be simplified by using the `union` function, such as it was done in the other implementations. Between lines 32 and 34 all lists are cleared.

```
1   Void CalculateMetricFRiCs(Property aProperty,
2                             Class aClass,
3                             Results aResults,
4                             List aBehaviorDiagramsList):
5     let vTmpList = newList():
6     let vClassesChildrenList = CollectAllChildClasses(aClass,
7                                                       aProperty,
8                                                       aResults.resultsList.size):
9     let vClassesRelatedList = CollectRelatedClasses(aClass,
10                                                    vTmpList,
11                                                    aResults.resultsList.size,
12                                                    vClassesChildrenList):
13    aResults.resultsList.add(CalculateMetricFRiC(aProperty,
14                                                 aClass,
15                                                 aClass,
16                                                 aBehaviorDiagramsList,
```

```
17                                          aResults.resultsList.size))−>
18   vClassesChildrenList.typeSelect(Class).select(c |
19     aResults.resultsList.add(CalculateMetricFRiC(aProperty,
20                                          aClass,
21                                          c,
22                                          aBehaviorDiagramsList,
23                                          aResults.resultsList.size)))−>
24   vClassesRelatedList.typeSelect(Class).select(e |
25       vClassesChildrenList.contains(e) != true &&
26       aResults.resultsList.add(
27         CalculateMetricFRiC(aProperty,
28                             aClass,
29                             e,
30                             aBehaviorDiagramsList,
31                             aResults.resultsList.size))== true)−>
32   vClassesRelatedList.removeAll(vClassesRelatedList)−>
33   vClassesChildrenList.removeAll(vClassesChildrenList)−>
34   vTmpList.removeAll(vTmpList);
35
36 create MetricResult CalculateMetricFRiC(Property aProperty,
37                                         Class aClass,
38                                         Class aRelatedClass,
39                                         List aBehaviorDiagramsList,
40                                         Integer aSize):
41
42   setId(5)−>
43   setText('Calculate Field Related in Class (FRiC)')−>
44   setTargetObject(aProperty.qualifiedName)−>
45   setTargetObjectType(Property.name)−>
46   setParentObject(aClass.qualifiedName )−>
47   setParentObjectType(aProperty.owner.metaType.toString())−>
48   setType(MetricsTypes::mtFieldRelatedInClass)−>
49   this.metricResultItem.add(CreateMetricItem(aRelatedClass,aSize))−>
50   setValue(CountRefBDFound(aProperty,aRelatedClass,aBehaviorDiagramsList));
```

Listing 4.21: Xtend `CalculateMetricFRiCs` and `CalculateMetricFRiC` functions

The function in Listing 4.22, implements the same routine as the `helper` `CountRefBDFound` in ATL Listing 4.4, the ETL `Operation` in Listing 4.10, and the `query` `CountRefBDFound` in QVT Listing 4.16 . In Xtend, differently from the ATL and ETL implementation, the function does not have a context, so that the field was passed as parameter (`aProperty`). Another obvious difference between the four implementations is the syntax.

```
1  Real CountRefBDFound(Property aProperty,
2                       Class aRelatedClass,
3                       List aBehaviorDiagramsList):
4    GetPropertyReferencesInOpaqueActions(
5                                         aProperty,
6                                         aBehaviorDiagramsList.typeSelect(Behavior).select(e |
7                                           e.owner == aRelatedClass),
8                                         aRelatedClass.qualifiedName) +
9    GetPropertyReferencesInCreateObjectAction(
10                                        aProperty,
11                                        aBehaviorDiagramsList.typeSelect(Behavior).select(e |
12                                            e.owner == aRelatedClass),
13                                        aRelatedClass.qualifiedName);
```

Listing 4.22: Xtend `CountRefDBFound` function

Listings 4.23 and 4.24 are very similar to the ATL, and ETL implementations. It is again only a syntax issue, since the queries are implemented exactly in the same way. One relevant difference can be seen in line 8 from Listing 4.5. To find a *substring* in a *string* in Xtend, the function `contains` must be used, differently from all languages analyzed before. Another difference is the use of the function `typeSelect` to typecast also unique in this implementation.

```
1 private Real GetPropertyReferencesInOpaqueActions(Property aProperty,
2                                                   List aBehaviorsDiagramsList,
3                                                   String aQualifiedName):
4   aBehaviorsDiagramsList.typeSelect(Activity).node.typeSelect(OpaqueAction).
5     input.typeSelect(ValuePin).select(e | e.name == aProperty.class.name).
6       value.typeSelect(OpaqueExpression).select(e |
7         e.body.first().contains(aProperty.name) == true).size;
```

Listing 4.23: Xtend `GetPropertyReferencesInOpaqueActions` function

```
1 private Real GetPropertyReferencesInCreateObjectAction(Property aProperty,
2                                                        List aBehaviorsDiagramsList,
3                                                        String aQualifiedName):
4   aBehaviorsDiagramsList.typeSelect(Activity).node.
5     typeSelect(CreateObjectAction).select(e | e.name == aProperty.name &&
6       && e.classifier == aProperty.owner).size;
```

Listing 4.24: Xtend `GetPropertyReferencesInCreateObjectAction` function

5. Language Evaluations

The four languages evaluations are divided in two analyses. One is based on the ISO 9126 (section 2.7) for evaluating the characteristics and quality of the tooling provided by the technologies, and by evaluating how these tools can contribute in enhancing the implementations quality. The evaluation approach of the languages continuity has been included in this thesis to evaluate characteristics related to the technology lifecycle. Predicting a technology lifecycle is fundamental in choosing it to be used in a project. This evaluation is based on the community use of the technology, technologies management, companies that supports the technology and its influence in the Information Technology (IT) market, how it is managed, its bug tracking system evaluation, and learning material and simple projects for helping new developers.

The second analysis is based on a study (section 2.8) for evaluating model transformation approaches [89] is used for evaluating the characteristics of the transformation language itself, and how this characteristics contributes to a better quality of the implementations.

Combining this two analyses it is possible to evaluate the transformation languages in a much broader way, since not only the languages characteristics themselves are analyzed, but also other factors important for the use of a technology such as, portability issues, learning material available, and project coordination analysis.

This chapter is structured in section 5.1, where the subjects of the analyzed technology are described, section 5.2, where the evaluation schema used is briefly described. In section 5.3 is the transformation language ATL evaluation described, section 5.4 describes the evaluation of the ETL transformation language, in section 5.5 is the evaluation of the Operational QVT transformation language described, and section 5.6 describes the evaluation of the Xtend transformation language.

5.1. Analyzed Subjects of the Transformation Language

The evaluation of the transformation languages is applied on the transformation language *tooling*, the *tooling implementation*, its *language features*, and the *transformation projects*. In this evaluation, the *tooling* is considered the set of tools provided by the technology that supports the development and execution of M2M transformations using the evaluated transformation language. One example of such a tool is a debugger, a *tooling* feature that enhances the *maintainability* of the *transformation project* by providing the user the feature of

better analyzing a bug by setting breakpoints, or analyzing the source code variables during the transformation execution. Another example is the use of the live error detection in the editor tool. This *tooling* feature provides a feedback for the user, displaying what part of the source code was erroneous implemented, and so enhancing the *usability* of the *transformation language*.

The *tooling implementation* is the source code of the technology *tooling*. The *tooling implementations*, and its coordination are important in the evaluation of its *continuity*, and ISO 9126 characteristics such as *maintainability* of the *tooling* provided. By analyzing the bug tracking system from the *tooling implementation*, it is possible to evaluate characteristics related to the *maintainability* of its *tooling*, such as evaluating the increase of bugs during the years, or by evaluating the amount of open bugs.

The *language features* are all features provided by the transformation language specification. Examples of such, as *inheritance* of rules, a feature that enhances the *transformation projects maintainability*, by reducing the amount of code of the implementation, or the possibility of executing *exogenous* M2M transformations, a feature that enhances the *transformation projects functionalities* by providing it with the feature of executing transformations between two different types of metamodels, such as UML and Ecore.

The *transformation projects* are projects containing implementations using the evaluated transformation language, such as a M2M transformation project for generating an UML model based on an Ecore model using ATL. A large amount of *transformation projects* enhances the *usability* of the *language features* by providing examples how to execute M2M transformations using a specific technology. *Transformation projects* can also impact in the *continuity* of the evaluated technology. A large set of *transformation projects* that provides many different examples showing and explaining the technology features can contribute to the decision making of the developer that wants to adopt a technology for M2M transformations.

5.2. Evaluation Schema

The characteristics *usability*, *efficiency*, *maintainability*, *functionality*, and *portability* from the ISO 9126 quality model, as also the *continuity* classification introduced in this thesis are used in the evaluation schema.

Table 5.1 shows the evaluation schema used in the evaluation of the analyzed transformation languages based on section 2.7, considering the subjects described on section 5.1. It is divided in two phases, the *implementation* phase, that are topics related to the implementation of the case study for each language, and the research phase, based on data collected from books, articles, and internet research.

Table 5.1.: Evaluation Schema

Topic description	**Phase**
Time needed to execute the compiled implementation.	Implementation
Configuring and compiling a template project without errors.	Implementation
Number of the case study implementation lines (LLOC metric)	Implementation
Time needed for the implementation of the case study.	Implementation
Tools support.	Research
Amount of public transformation projects available.	Research
Metric calculation and bad smell detection public projects found.	Research
Different technologies that may be used in conjunction.	Research
Number of manufactures that officially support the use of this technology.	Research
Group responsible for the technology maintenance and coordination.	Research
Release frequency in the last two years.	Research
Last official release containing new features.	Research
Bug tracking system analysis	Research
Amount of books not older then 2 years found in 20 minutes research.	Research
Amount of articles not older then 2 years found in 20 minutes research.	Research
Amount of tutorials found not older then 2 years in 20 minutes in research.	Research
Learning material and information about the technology provided by the official website.	Research
Table with the quantity of topics and replies in the official forum from the last five years.	Research
Quantity of topics and replies in the official forum from the last five years.	Research

The time needed to execute the implemented code is measured to analyze the technology's *efficiency*. It is measured the time needed to execute the implementation from the case

study. The implementation's source code time is the average time of executing the implementation ten times[1]. It is important to observe that the source model is small compared to models used in the industry, and therefore cannot be considered an empirical evidence.

The topics from the evaluation schema in Table 5.1, are referenced in the schema evaluation from each language in **bold** letters. In the evaluation of the language characteristics, the keywords defined in the study from Czarnecki and Helsen are written in *italic*. Considering the conclusion from each language, the *slanted* text formatting represents the ISO 9126 main characteristics, including the introduced continuity classification. The **bold** texts refer to the technology subjects, introduced in section 5.1. The highlighting of these keywords should enhance the evaluation understanding.

5.3. ATLAS Transformation Language (ATL)

In the sections below is evaluated the ATL transformation language.

5.3.1. Schema Evaluation

The **time needed to execute the compiled implementation** in ATL depends on the virtual machine that is being used in the compilation. It is possible to use two different Virtual Machines. The *Regular VM* that is the first version of the ATL VM, in which model handlers are used to abstract the implementation from the model management framework. This model handlers are an abstraction layer that are dedicated to model access, and they are implemented in two different classes, the `ASMModel` and `ASMModelElement`.

According to [12], the VM is still in use because it is strongly linked to several parts (now only the ATL debugger). This source also implies that the *Regular VM* also has several performance issues, especially because of the model handle architecture. The other VM is the *EMF-specific VM*, that according to [12], is a redefinition of the *Regular VM*, which resolves a lot of performance issues by avoiding `EObjects` wrapping. Its API also allows to consider EMF resources directly as models.

It was possible to observe this performance issues in the case study implementation. The *Regular VM* took 1,43 seconds[2] to execute the `UML2BSDMC_MM_p1.atl` file, and 3,537 seconds to execute the `UML2BSDMC_MM_p2.atl` The total time needed to execute the complete case study was 4,974 seconds. The *EMF-specific VM* took only 0,325 seconds to execute the `UML2BSDMC_MM_p1.atl` file, and 0,459 seconds to execute the `UML2BSDMC_MM_p2.atl` file, with an overall of 0,784 seconds to execute the entire case study. It is possible to observe

[1]Computer configuration used is a Macbook Pro, 2.66 GHz Intel Core 2 Duo, 4 GB 1067 MHz DDR3 RAM, Mac OS X Snow Leopard 10.6.2 operational system, and Eclipse IDE, build 20090920-117.

[2]This value has been calculated by taking the average time of 10 consecutive executions.

that the *EMF-specific VM* was more than 6 times faster than the *Regular VM* for our case study.

Configuring and compiling a template project without errors was a straightforward process. The *Eclipse Modeling Tools* IDE standard installation provides all necessary packages, plugins and tools necessary to execute and write ATL code. Having a project with all metamodels and models referenced, output file defined and one rule to test the output file took approximately 30 Minutes. A Graphic User Interface (GUI) was also available for setting up the metamodels, models, libraries, superimposed modules, and the VM used in the execution.

The **number of the case study implementation lines**, not considering the source code documentation is 618 lines. The queries implemented in ATL are not as compact as in other languages, they are described in the section 5.3.3. It was necessary **approximately 70 hours to implement the entire case study.** The ATL **tool support** is integrated to the Eclipse IDE, providing the user with code completion, syntax highlighting, some level of error detection in the source code, an automated build tool using Apache Ant, debugging functionalities, outline view, GUI for configuring the *run* and *debug* project function in Eclipse.

The **amount of public transformation projects available** for ATL is 103 transformation scenarios, a few examples are *Ant to Maven* transformation, *UML to Java* transformation, and *MOF to UML* transformations, all found on the official ATL website.

The **metric calculation and bad smell detection public project found** was one. This transformation project named *UML2 to Measure* can be found in [9]. The available metric calculations in this transformation project are QMOOD, MOOSE, FLAME, and MOOD.

Different technologies that may be used in conjunction with ATL are OCL, Apache Ant, ATLAS MegaModel Management (AM3), AMW, Java, and also the technologies supported by the EMF.

The OCL language is described in chapter 3.5.1. According to [78], ATL is designed and implemented with respect to the OCL standard. There are also a few derivations from the standard, and they are also described in [78]. The Apache Ant is a Java-based build tool. By using Apache Ant, it is possible to create workflows with ATL, making the execution of ATL large projects much easier to manage. The AM3 is a GMT subproject witch is being developed by the *AtlanMod* group, that is same group that coordinates the ATL project. The AM3 has a group of Apache Ant tasks that can be used in ATL transformations. It is also possible to use AMW, that was also developed by the *AtlanMod* group, and can be used for model traceability. It is also possible to run ATL transformations using Java code. A Java application template project can be found in [14]. The EMF is a framework that integrates a larger set of technologies that can be used in conjunction. The EMF is available in the *Eclipse Modeling Tools*, more information about it can be found in [144].

The ATL **number of manufactures that officially support the use of this technology** can be found in [10]. Nine different partners are listed, including companies such as Airbus [1], and Thales [63].

The **group responsible for the technology maintenance and coordination** is *INRIA*, more specifically the *AtlanMod* group, that is located in Nantes, France. The *AtlanMod* group is also known for the AtlanMod Model Management Architecture (AmmA) toolkit, and also participates in other projects in the MDE field such as Carroll [17], ModelPlex [45], TopCased [66], OpenEmbeDD [52], OpenDevFactory [68], FLFS [54], IdM++ [60]. There are eight active members according to [43].

In Table 5.2, it is possible to observe the **release frequency in the last two years**.

Table 5.2.: ATL releases in the last two years

Version	Date
3.0.1	22.09.2009
3.0.0	22.06.2009
2.0.2	19.12.2008
2.0.1	17.09.2008
2.0.0	10.06.2008

The **last official release containing new features** at the time of writing is version 3.0.1. The **Bug tracking system analysis** in Eclipse Bugzilla in the last five years[3], filtering the project M2M and components *ATL-Contribution*, *ATL-Doc*, *ATL-emfvm*, *ATL-Engine*, *ATL-UI*, and *ATL-Website* are shown in Figure 5.1 and Figure 5.2. No bug entries were found from the period of 2005 to 2006 in the Eclipse Bugzilla system.

[3]2009 until 08.12.2009.

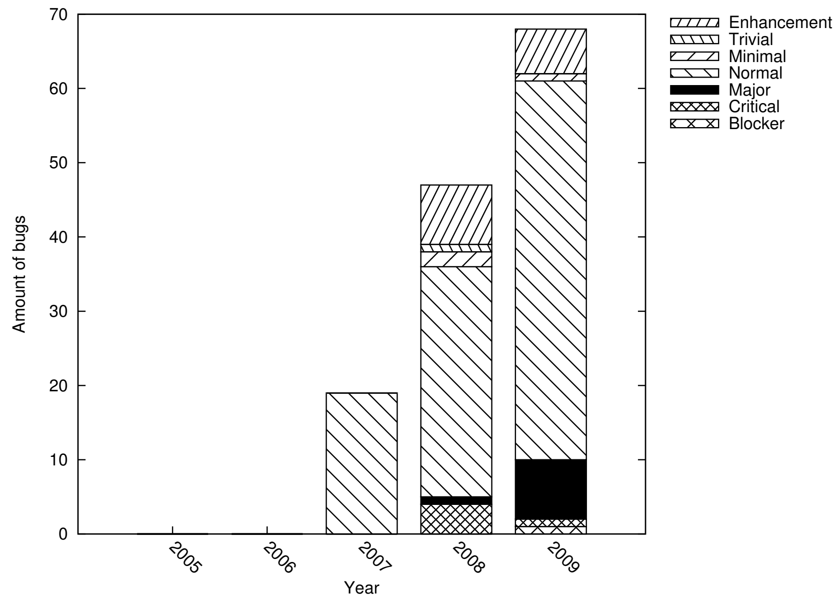

Figure 5.1.: Eclipse Bugzilla ATL projects entries grouped by severity.

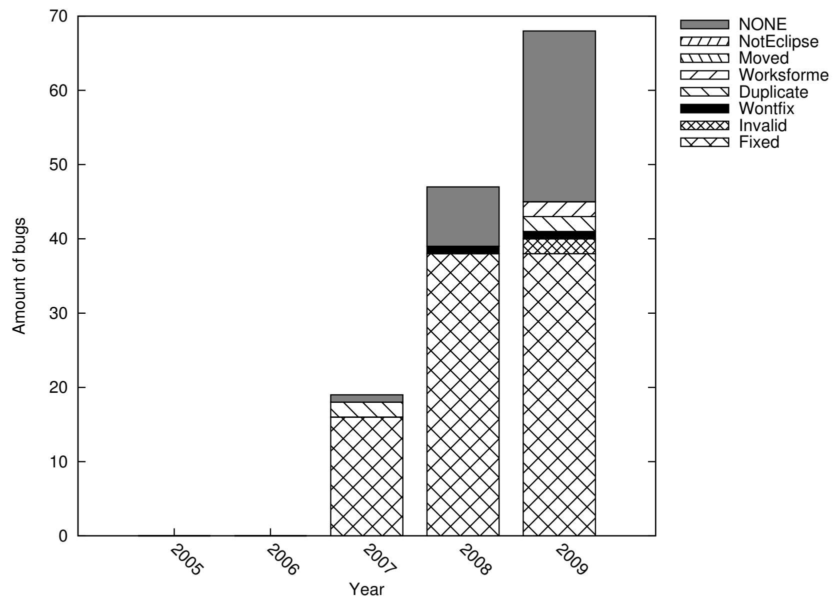

Figure 5.2.: Eclipse Bugzilla ATL projects entries grouped by resolution.

In the search for the **amount of books not older then 2 years found in 20 minutes research**, it was possible to find [101]. But this book is not ATL specific. The **amount of articles not older then 2 years found in 20 minutes research** found is 18. It can be found in [86, 92, 111, 120, 136]. [111] contains a large set of articles. The **amount of tutorials not older then 2 years old in 20 minutes research** found is 3 and can be found in [5]. The **learning material and information about the technology provided in the official website** found is composed by several examples, from basic to complex ones, a detailed documentation for users and developers, the virtual machine specifications, installation guide, starter's guide, wiki, newsgroup, forum, Frequently Asked Questions (FAQ), links to publications, posters, and flyers. Figure 5.3 shows the **quantity of topics and replies in the official form from the last five years**[4].

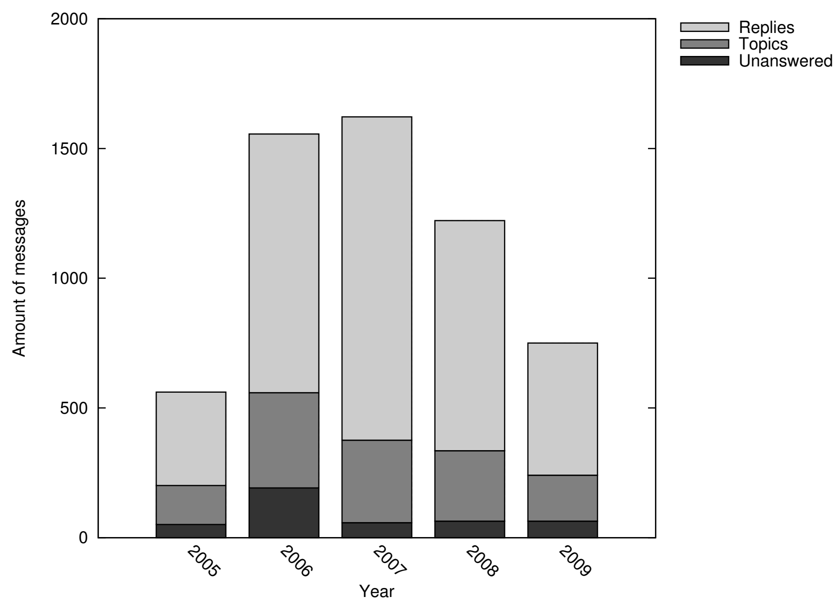

Figure 5.3.: ATL forum usage.

The period of 2005 to 2006 is based on the data collected in [13]. The period of 2006 to 2009 has been collected in [25], but in this case, since this forum is used for ATL and QVT, the messages have been filtered by the string ATL in the subject and in the message content. It is possible to observe that the activity peak was in the year of 2007, and since then have been declining in the last years. The **quantity of topics and replies in the official forum from the last five years** is 5710.

[4]2009 until 02.12.2009.

5.3.2. Language Characteristics

The ATL transformation language is evaluated based on [89] is described below.

Specification

ATL provides a feature of calling explicitly the *entry point* rule before calling all other matched rules, this *entry point* rule can be used for initializing variables, executing one imperative block, and also for analyzing *pre* conditions from the model transformation.

Transformation Rules

ATL *transformation rules* provides a *syntactic separation* between the target and source models, that are achieved by using the `to` and `from` keywords. It also provides the *application controls* feature by creating conditions inside the `from` block for defining the rules condition for its execution. The ATL rules does not support *multidirectionality*, therefore transformation is always performed in one direction. According to [109], *intermediate structures* are supported via attribute helpers and OCL tuples. We do not consider in our evaluation OCL tuples as *intermediate structure*, since they are part of the target or source model. But using a second target model for tracing the M2M transformation, and saving it at the end is a way of using it as an *intermediate feature* is in ATL. Rules *parameterization* are only supported in called rules, and not in matched rules. According to [89], *reflection* is supported in ATL by allowing reflective access to transformation rules during the execution of the transformation. The *domain languages* supported by ATL are Ecore and MOF. The M2M transformations can be *endogenous*, or *exogenous*. Since ATL transformation is unidirectional, the only *static modes* supported are *in*, and *out*. It does not support *dynamic mode restriction*. The *body* supports *variables*, *patterns*, and its executable *logic* is based on a hybrid *language paradigm*, that supports imperative and declarative implementations, the *value specification* is *imperative assignment*, and the *element creation* is *implicit*. ATL is considered *syntactically typed*.

Rule Application Control

The *rule location determination* is *deterministic* in the case of called rules, and *non-deterministic* in the case of matched rules according to [109] .

The *rule scheduling form* is a mix of *implicit* supported by matched rules, and *explicit* rules supported through the invocation of called rules, and also through control flow structs. The *rule selection* considering the matched rules are based on the rules guard conditions. According to [109], conflicts among standard matched rules are not allowed when they lead to ambiguity in resolving the default traceability links. Concerning the *rule interaction*, ATL provides rule *recursion*. *Phasing* can be used by calling explicitly from inside *matched*

rules a specific order of *lazy rules*, such as calling a specific sequence of *lazy* rules from an *entry point* rule. In this way it is possible to use *phasing* with *lazy* rules.

Rule Organization

Concerning the *rules organization* in ATL, its *modularity mechanism* consists in organizing the rules in libraries and modules, its *reuse mechanism* is based in rule *inheritance*, and also module *superimposition*. Its *organizational structure* is *source-oriented*.

Source-Target Relationship

The *Source-Target relationship* is that a *new target* model is created in the transformation. According to [89], *in-place* transformation can be simulated in ATL through an automatic copy mechanism.

Incrementality

ATL does not support any type of *incrementality*. In a transformation process, the entire source model is read, and the target model is written, so that changes performed by the user in the target model are cleared in a new transformation execution.

Directionality

The ATL transformation *directionality* is *unidirectional*, from the source to the target model.

Tracing

The *tracing* support is provided by ATL.

5.3.3. Further Comments

The implementation took more time than compared to other technologies. The possibility of using *matched*, *lazy*, and *called* rules, gives the developer a lot of flexibility in the implementation, such as an *entrypoint* rule that can be used for initializing variables. In our point of view, ATL should also provide the possibility of executing one specific code at the end of the transformation, such as it is done in Epsilon. Our case study first calculates all metrics, and based on those metrics, the bad smells are calculated. It was not possible to call the bad smell detection in the `UML2BSDMC_MM_p1.atl` implementation, since it does not provides a `post` block, such as in Epsilon. It was necessary create a second transformation `UML2BSDMC_MM_p2.atl` to detect the bad smells.

There was also an issue in the automated build tool Apache Ant. It was not possible to execute the Apache Ant file in the case that superimposed files were defined in the Apache Ant file. Another interesting issue is in the comparison of `integer` and `real` values. In the example below (Listing 5.1), on line 9 it is possible to observe that the field `value` from the object `aRelatedMetricResults` is being compared. In the case that the `value` field is the from primitive type `real`, it must be compared such as in the example using `0.0`. But in cases that the `value` is from the `integer` type, then it must be compared with `0` and not `0.0`. If those values are not compared as described before, no error is generated in the execution, but the comparison is not resolved correctly.

```
1 helper context QueryResult!Results def :
2   IsFiWC(aRelatedMetricResults : QueryResult!MetricResult,
3            aPropertyName : String,
4            aParentName : String) : Boolean =
5     (aRelatedMetricResults.MetricResultItem−>first().relatedObject <>
6           aPropertyName and
7      aRelatedMetricResults.MetricResultItem−>first().relatedParentObject <>
8           aParentName and
9      aRelatedMetricResults.value <> 0.0);
```

Listing 5.1: ATL comparison issue

Another observation in the ATL implementation, is that queries written in ATL are much longer than written in, for example, QVT. Observing the Listing 5.2 and 5.3, it is possible to observe in the Listing 5.3 on line 1, that all behavior diagrams from the type `Activity` are filtered and collected by using the instruction `[Activity]`. To get exactly the same set in the ATL implementation, it is necessary to write an entire `select` to get the same result. In this case it would be `->select(e|e.oclIsTypeOf(Source!Activity))`. In the example it is also possible to observe that ATL must convert the `size()` value, that is an `integer` value, to a `real` value, while QVT accepts `integer` values for `real` fields.

```
1 aBehaviorDiagramsList−>select(e |
2   e.oclIsTypeOf(Source!Activity))−>collect(f | f.node)−>flatten()−>
3     select(g | g.oclIsTypeOf(Source!CallOperationAction))−>select(h |
4       h.operation = self)−>size()−>toString()−>toReal()
```

Listing 5.2: ATL query example

```
1 aBehaviorDiagramsList[Activity].node[uml::CallOperationAction]−>
2   select(e | e.operation = self)−>size();
```

Listing 5.3: QVT query example

The current language implementation also allows the user of calling lazy rules with less parameters than it is declared, may causing some unexpected results in the transformation execution.

The syntax autocompletion also had a few issues in showing all available operations and fields from elements in some specific cases.

The debugger functionality is only available in the *Regular VM* , and not in the *EMF-specific VM*, the *Variables* view in the *Debugger* view also had some issues in displaying the correct variable values. In our implementation, it only showed the objects number and did not display all other properties.

The rules inheritance feature is available in the *ATL2006* compiler. The *ATL2006* does not support multiple inheritances, and there are a few limitations and constraints that should be taken in consideration before using inheritance, they are described in [6] and are:

- sub rules input pattern has to match a subset of its super rule.
- input pattern variables names have to be the same in super and sub rules.
- output pattern variables names in the sub and its super rule must be equal.

Another characteristics of the rules inheritance behavior, is that only the `from` block from superclasses are executed, all other blocks are ignored, differently from the QVT implementation, where all blocks are executed.

To write one enumerated type in ATL it is necessary to write the character # before writing the enumerated type value, such as for example `#public`. Another transformation languages such as Xtend for instance, uses the following `uml::VisibilityKind::public` notation, which enhances the understandability of the source code.

5.3.4. Conclusions

The conclusions described below are based on the evaluation schema, transformation language classification, and on the case study implementation. Each characteristic is evaluated individually.

Efficiency

Concerning the technology *efficiency* of the **tooling** provided by the technology, the time needed for executing the case study implementation is the fastest one between the evaluated technologies. Executing the case study implementation using the *EMF-specific VM*, it is more than twice as fast as the second fastest transformation language that was QVT. The high performances of the technology in executing source codes is very important in tasks such as detecting bad smells, calculating metrics, performing refactoring, and execute complex transformations in large models.

Portability

All the **tooling** provided by ATL is available in the Eclipse Modeling Tools IDE standard installation, so that it was not necessary to install any plugin, or library to configure the

ATL **tooling**. Considering the **transformation projects**, creating a new project, and executing it without errors was a fast compared to other transformation languages. A few examples of transformation projects available in the official site where installed for testing, without finding any kind of *portability* problems. No problems related to the *portability* of its **tooling** and its **transformation projects** have been found.

Usability

Considering the *usability* of its **tooling** and **language features**, it was the transformation language that had the highest implementation time, almost two times more compared to the language with the fastest implementation time. Concerning the *usability* of its **tooling**, it was not possible to use the code autocompletion from the metamodels used in the transformation, to better understand how the structure is defined. The autocompletion of metamodels in the editor enhances the *usability* of the **transformation projects** source code. Concerning the technology **language features**, more time was also necessary to understand how the rules are executed, and how to implement the case study keeping some level of similarity to other transformation languages. The *tracing* feature provided by its **tooling**, that tracks and saves in a file the process of a M2M transformation, enhances the understandability of a M2M **transformation project** execution. The amount of publicly available third party **transformation projects** is very good, not only quantitatively, but also qualitatively. It is the only technology among the ones evaluated, that provides a transformation project for calculating metrics, providing a good source of information in tasks related to UML quality engineering in metric calculations. It provides examples of transformations of a good amount of different domains, which is a good material for learning and understanding its **tooling**, **language features** and the **transformation projects** itself.

Technologies which are not ATL specific, that can also be used for other tasks, such as Apache Ant, integration to EMF, and Java, make the understanding and learning of its **tooling**, and **transformation projects** faster, since the developer do not have to learn technology specific tools, and plugins. Another factor that also contributes to a better understanding and learning of the **language features** is the support of OCL standard, that is also used in other technologies.

The books, articles, and tutorials found in the 20 minutes research are considered good. They provide a good source of information for learning and understanding its **tooling**, **language features**, and **tools implementations**. It was possible to find two books, but none of them are specialized in ATL. A good amount of articles was also found in the research. The Learning material found in the official ATL project is good organized and up-to-date. The ATL forum (Figure 5.3) is a good source of learning material and contact to the ATL community. It is possible to observe that it has a high amount of messages compared to the other transformation languages forums, but proportionally, it has a higher amount of unanswered topics compared to the other evaluated languages.

Maintainability

Concerning the *maintainability* of the **transformation projects**, the ATL **tooling** provides a *tracing* feature for analyzing M2M transformation processes, a debugger that is especially useful in analyzing long OCL-like expressions in large models. Issues related to the debugger tool were found and it is described in section 5.3.3. A refactoring tool is useful in the *maintainability* of **transformation projects**, but it is not available for the ATL editor. Considering the implementation of the case study, the ATL transformation language has the highest amount of source code lines, therefore not contributing to a good *maintainability* of the **transformation projects**, since a high amount of source code lines can imply a higher *maintainability*. Table 5.2 shows the releases in the last two years. The older release does not only contain bug fixing, but also enhancements. This is a good indicative that the ATL **tooling implementation** is been *maintained*. In the ATL version 3.0.0 the APIs was refactored, as also a few modifications in the routine for executing in-place transformations were done. The release interval between versions 2.0.2 and 3.0.0 is two times longer than the usual interval release. A relatively constant release frequency is a good indicative of a coordinated *maintainability* of its **tooling implementation**. In version 3.0.0 ATL now supports its own set of Apache Ant tasks, and other features available [4], so that this implementations could contribute to the longer development time between those releases.

The *maintainability* of the ATL **tooling**, **tooling implementation**, and **language features** can be evaluated by analyzing its bug tracking system entries. The bug entries found in the bug tracking system contains information related the quality of the *maintainability* of its **tooling implementation**, as also enhancements request from the community related to its **tooling** and its **language features**. It is possible to identify in Figure 5.1 that the amount of entries is increasing. A few factors could contribute to the increase of bug entries, the community using the technology is growing, so that that its **tooling** is being used more often, therefore the possibility of finding bugs is higher. Another factor that could contribute to the increase of bugs are quality issues in its *maintainability*. Yet another factor that could contribute to an increase of bugs in this year is the release of the major version 3.0.0. Another observation is the bug severity distribution in 2007 compared to the other years. The grand majority of bugs in 2007 are normal bugs, while in the other years the distinction between bugs severity is made, so that it could indicate a change of policies in the bug severity classification by the coordinating group. A good policy of the bug classification contributes to a better *maintainability* of its **tooling implementation**, by defining the most important bugs by severity. Figure 5.2 shows the amount of bugs found, filtered by status. It is possible to observe that the amount of open bugs has increased in 2009, but the amount of fixed bugs are very similar to the previous year, so that it could indicate that an insufficient number of developers is working in the ATL project, which is not a good indicative of *maintainability*. The ATL **language features** support the use of rules *inheritance*, *superimposed* module, and rules organization in modules, features that also contributes to

the *maintainability* of *ATL* **transformation projects**.

Functionality

Considering the technologies that can be used in conjunction, ATL **tooling** supports the integration with Apache Ant for automating tasks, AM3 provides Apache Ant tasks for ATL, and AMW can be used for tracing. This integration enhances the *functionality* of the technology and its **tooling**. Java implementations can be called from ATL transformation executions, or standalone Java applications that execute transformations using ATL, enhances the **transformation projects** *functionalities*.

It was possible to find one transformation project for calculating metrics in models. This and other projects enhance the *functionality* of **transformation projects**, since many of the functions and libraries already implemented in **transformation projects** can be reused. This project for calculating metrics in models, has a set of libraries for calculating metrics from the QMOOD, MOOSE, FLAME, and MOOD groups of metrics. This project makes a M2M transformation, analyzing the source model, and generating a target model containing the calculated metrics results. This approach is similar to the approach adopted in the case study, but it is limited to calculating metrics. The libraries provided in this project can be used in our case study to expand its *functionalities* in calculating metrics. A good amount of other transformation projects are available in the official website, so that parts of its implementations can be used in other projects.

Concerning the *functionality* provided by the ATL **tooling**, it provides a GUI for configuring the transformation execution, the editor supports almost all features provided by the editors from the other examined languages, but its code autocompletion, and live error detection is not so advanced, such as editors found in other transformation languages. These *functionalities* provided by the **tooling** can enhance the *maintainability* of **transformation projects**.

Considering the *functionalities* provided by the **language features**, the *phasing* characteristic used in UML quality engineering can be used in bad smell detection process, in which the first phase is used to calculate the source model metrics, and the second phase is used to calculate the bad smells based on those metrics. It was the only language in our case study where it was necessary to execute two transformation processes in a row, one for calculating the metrics, and the other for detecting the bad smells. It also simulates *in-place* transformations that can be used in refactoring processes. It also supports exogenous transformations, that is the transformation of a source model that conforms to a specific metamodel to a target model that conforms to a different metamodel. Intermediate structures are also supported by using more then one output model.

Continuity

The large quantity of publicly available third party **transformation projects** in ATL provides a good source of information for developers that are evaluating M2M transformation approaches before choosing one, so that a considerable amount of this **transformation projects** contributes to the adoption of this technology in other projects. In our evaluation ATL provided a very good set of **transformation projects**, not only quantitatively but also qualitatively.

The number of manufactures that support the technology is an important factor in the decision making of adopting one technology. By analyzing the companies that support this technology, it is possible to observe that many of these manufactures, or their headquarters are located in France. There is a good set of companies that provide training and consulting, two important factors that contribute in the adoption of a technology. Two large companies that support the ATL projects are Thales, and Airbus. The ATL project is coordinated by the *AtlanMod* group, that is composed by eight active members. This group is also active in other MDA projects listed in [55]. Based on this information it is possible to observe that this is an active group, with experience in other MDA projects.

The release frequency compared to other transformation languages is considered good, and the last version also contains new features, a good indicative of the technology *continuity*. Analyzing its documentation, it is possible to observe that enhancements are also being implemented, that shows that the technology is continuing to evolve, another good indicative of *continuity*.

It is possible to observe a growth of bug entries in the bug tracking system Bugzilla, which could indicate a growth of the community using ATL. Figure 5.3 shows the forum activity in the last five years, compared to the other transformation languages, is currently one of the most active in posts and replies. It is possible to observe that the number of messages is decreasing in the last two years, this could indicate that the technology is not being used so much as before, or that the material found by the community about ATL is self explanatory. It is also relevant remark that the *INRIA* research group, is also referenced as one of the companies that supports the QVT specification.

5.4. Epsilon Transformation Language (ETL)

ETL is one of seven different languages available in the Epsilon Eclipse project. A few of the topics evaluated are not specifically based on ETL, but on the Epsilon project. These topics are those related to project coordination, bug tracking system analysis, forum usage, tool support, and the material available in the official site. All other topics are evaluated exclusively taking in consideration only ETL language.

5.4.1. Schema Evaluation

The **time needed to execute the compiled implementation** in Epsilon was 4 seconds[5] For **configuring and compiling a template project without errors** it was necessary to install the Epsilon plugin in Eclipse, since it is not available in the default Eclipse Modeling Tools installation. It took approximately 120 Minutes to prepare a project with all metamodels, models, and libraries referenced, output file defined and one rule to test the output file. The **number of the case study implementation lines** is 574, also considering the workflow file, and excluding the source code comments. It was necessary **approximately 35 hours to implement the entire case study**.

The **tool support** provided by Epsilon is composed by five different tools described in section 2.9.2, the EuGENia tool is used to automatically generate *.gmfgraph*, *gmftool*, and *gmlmap* models needed for implementing a GMF editor from a single annotated Ecore metamodel. *Exeed* is a build-in EMF reflective tree-based editor. It can be used to customize the appearance of nodes in the reflective tree editor without having to generate a dedicated editor, it can specify icons and labels of each node using EOL, and this icons and labels can reflect the status of an element. The *ModelLink* is an editor that allow the user to display three different EMF tree based editors side-by-side. The *Workflow* is basically a set of Apache Ant tasks. *Concondance* is a tool used to monitor a group of projects from a workspace maintaining the EMF cross-references. The Epsilon editor provides code autocompletion, syntax highlighting, outline view, and traceability view. Unfortunately it does not have a debugger, and it does not provide live error detection in the editor. An interesting feature is the *Live* version available in [42], which allows the execution of EOL implementations from a web browser.

The **amount of public transformation projects available** found in its official website is 6, considering only the examples that uses ETL. There was no **metric calculation and bad smell detection public transformation projects** found.

The **different technologies that may be used in conjunction** with Epsilon ETL, are all Epsilon available languages, Apache Ant, Java, by running Epsilon transformations in a Java standalone implementation, use models and metamodels specified in EMF. Epsilon also provide experimental projects available in [59], between those projects is one driver projects for accessing Meta Data Repository (MDR) based models in Netbeans [70], and another one for accessing Z specification language [141] based models used in [20] from Epsilon.

Considering the **number of manufactures that officially support the use of this technology**, according to the leading developer, parts of Epsilon is used by people in companies including IBM (Zurich) [37], Siemens [56], BAE Systems [16], WesternGeco [34], and Telefónica [62] in a informal way, so that the companies do not officially support the Epsilon

[5]This value has been calculated by taking the average time of 10 consecutive executions.

project. It is also used in several European Union (EU) projects such as Ample [36], Iness [19], and Modelplex.

The **group responsible for the technology maintenance and coordination**, is coordinated by the leading developer Dimitris Kolovos [114]. According to the leading developer, two persons are officially committing changes in the Epsilon source code, but there are several other people also working in the project, and two new committers are expected to be added in a new future. The enhancements are defined based on the ideas and request from the community in the form of bugs or forum messages. An internal group discussion is made to define the enhancements that are going to be implemented.

In Table 5.3 it is possible to observe the **release frequency in the last two years**, considering only the stable versions. These version dates were found in the official Epsilon webpage. The Epsilon project was a standalone project until 17.06.2008. By integrating it in to the Eclipse modeling project, it was necessary to adapt the projects version to the Eclipse version structure. In Eclipse, every incubated project must start with version 0, therefore it is an incubated project, and it received the new version 0.8.0, such as it is shown in Table 5.3. The **last official release containing new features** is newest version at the time of writing.

Table 5.3.: Epsilon releases in the last two years

Version	Date	Version	Date
0.8.8	28.10.2009	0.8.1	12.09.2008
0.8.7	03.07.2009	0.8.0	17.06.2008
0.8.6	12.06.2009	1.3.6	17.06.2008
0.8.5	27.04.2009	1.3.5	21.05.2008
0.8.4	11.02.2009	1.3.4	20.05.2008
0.8.3	08.12.2008	1.3.3	18.03.2008
0.8.2.1	03.11.2008	1.3.2	14.03.2008
0.8.2	28.10.2008	1.3.0	29.01.2008
0.8.1.1	16.09.2008	1.2.0	13.12.2007

The **Bug tracking system analysis** in Eclipse Bugzilla in the last five years[6], filtering the project GMT and Epsilon component, and it is shown in Figures 5.4 and 5.5. No bug entries were found from the period from 2005 to 2006 in the Eclipse Bugzilla system.

[6] 2009 until 08.12.2009.

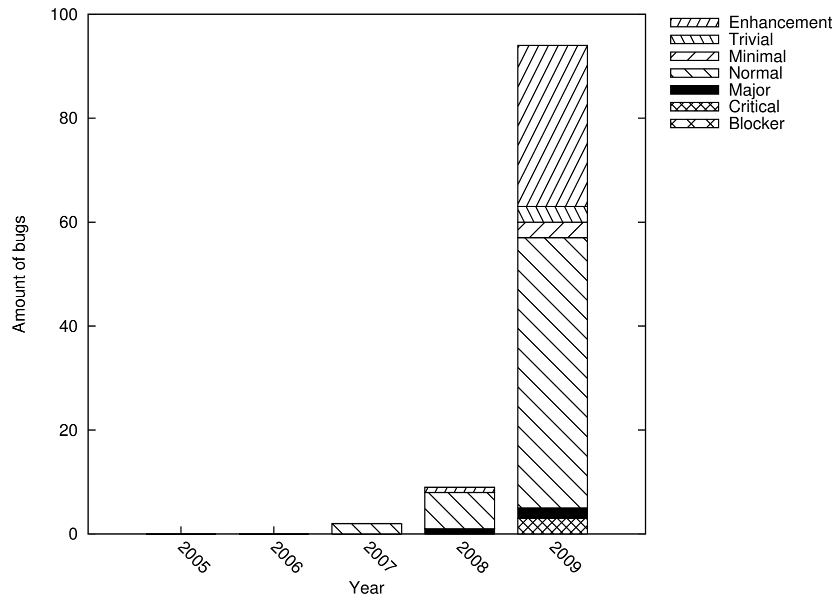

Figure 5.4.: Eclipse Bugzilla Epsilon projects entries grouped by severity.

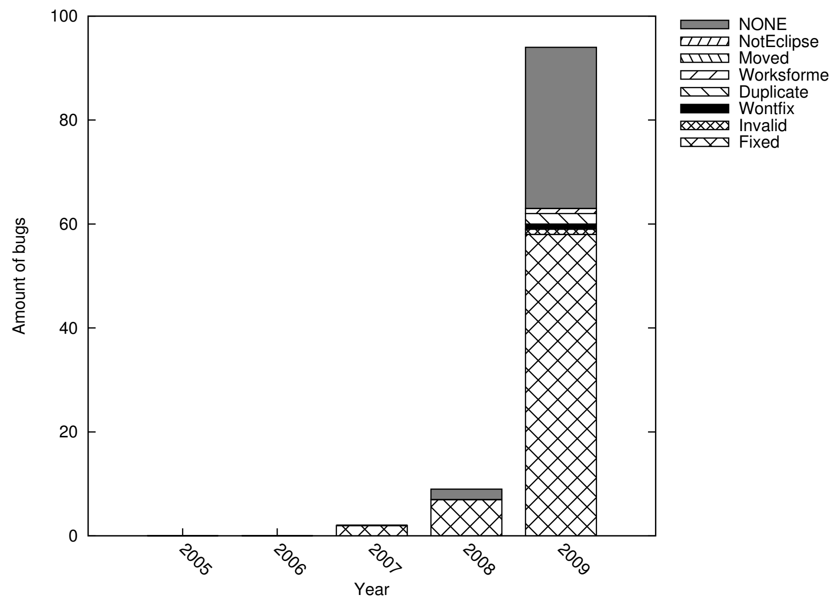

Figure 5.5.: Eclipse Bugzilla Epsilon projects entries grouped by resolution.

The **amount of books not older then 2 years found in 20 minutes research** is one. It was only possible to find the official Epsilon book [118]. The **amount of articles not older then 2 years found in 20 minutes research** is 10. Since Epsilon is composed by diverse languages, only articles that have at least one reference to the ETL language were considered. It was possible to observe that the majority of the articles found are from a small group of researches. Relevant articles found are [93, 116, 117, 137]. The **amount of tutorials found not older then 2 years in 20 minutes research**, considering only tutorials related to the ETL language, was none. It was possible to find screencasts, but none related to ETL. Considering the **learning material and information about the technology provided by the official website**, it was possible to find two simple examples of transformation using ETL, as also a good documentation from the Epsilon project in the form of an eBook [118]. It was also possible to find 12 articles, but none directly related to ETL. The official site also provides a FAQ, Web Log (BLOG), forum, newsgroup, and one live application for executing EOL programs. It was also possible to find 10 screencasts, but none related to ETL. The figure below shows the **quantity of topics and replies in the official form from the last five years.**[7]

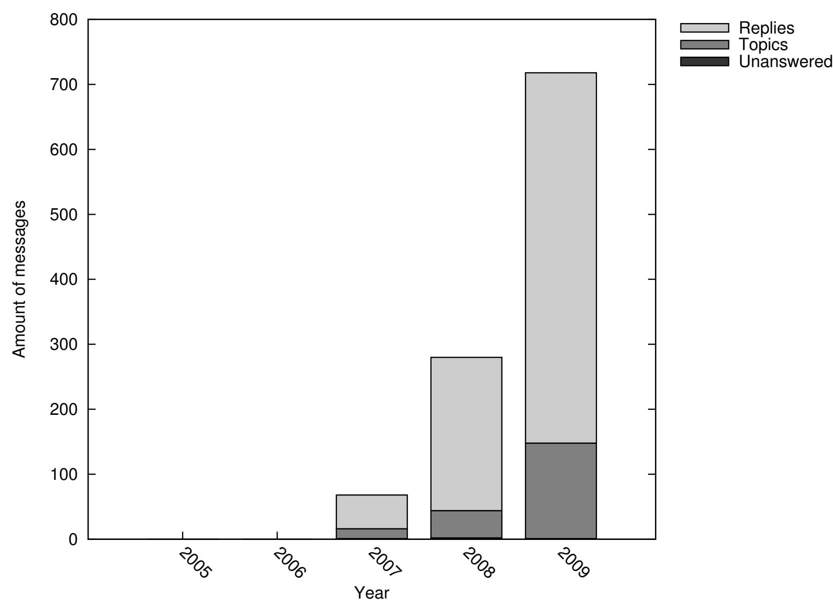

Figure 5.6.: Epsilon forum usage.

The Epsilon project was started in 2005, it was not possible to find a forum from the period of 2005 to 2006. The data collected in 2007 was based on the forum located in [24],

[7]2009 until 02.12.2009.

the period from 2008 contains data from the forums [23, 24] and 2009 from forum [23]. The **quantity of topics and replies in the official forum from the last five years** is 1066.

5.4.2. Language Characteristics

The ETL transformation language is evaluated based on [89] is described below.

Specification

Epsilon calls explicitly the `pre` block before calling all matched rules, and after the transformation the `post` block is executed. The `pre` block can be used for initializing variables, analyzing *pre* conditions, or one specific imperative implementation. The `post` block is executed at the end of the transformation process, and can be used to execute one imperative part of the implementation, such as it is done in our case study where the `post` block is used to detect the bad smells in the source model.

Transformation Rules

The *transformation rules* provides *syntactic separation* between the target and the source models. This separation is achieved by using the `transforms` and `to` keywords. In the `transforms` block is set the element that must match with the source model element, and the `to` block contains the target model element that is created in the case of a match. It also provides *application controls* feature by creating a conditions in the *guard* block for executing the rule or not. ETL also does not provide *multidirectionality*, so that transformations are executed from the source to the target model. According to [117], ETL supports the creation of more than one output model. We consider that this feature provides ETL with the *intermediate structure* feature, so that a second output model can be used during the transformation for other purposes, such as for tracing the M2M transformation. Rules in ETL do not support *parameterization*, only ETL `operations` does.

The *domain languages* supported by ETL are Ecore, and UML, and the transformations can be *endogenous*, or *exogenous*. The *domain static mode* supports *in*, *out*, and *in/out*, depending on how each domain is set in the workflow. It was not possible to find a way of using *dynamic mode restrictions* in ETL, since the *domain static modes* are defined before executing the transformation. One possible solution would be to execute more transformations in one workflow, and setting the *domains static modes* for each transformation differently. The *body* supports *variables*, *patterns*, and its executable *logic* is based on a hybrid *language paradigm*, the *value specification* is *imperative assignment*, and the *element creation* is *implicit* or *explicit*. ETL is considered *syntactically typed*.

Rule Application Control

The *rule location determination* in ETL is *deterministic*. By executing the ETL implementation in the case study it is possible to observe a deterministic way of accessing the elements in the source model.

The *rule scheduling form* is a mix of *implicit* supported by matched rules, and *explicit pre* and *post* blocks, that are called automatically by the transformation process. The *operation* blocks are also called *explicitly* from inside rules and other *operations*. The *rule interaction* supports *operations recursions*. *Phasing* is also possible by executing ETL transformations in a sequence, or using the *pre* and *post* blocks to explicitly execute a specific sequence of *operations*.

Rule Organization

Considering the *rule organization*, its *modularity mechanism* is provided by organizing the rules in modules. The *reuse mechanism* provided by ETL is the support of rule *inheritance*, and its *organizational structure* is *source-oriented*.

Source-Target Relationship

The *source-target relationship* provided in ETL is defined in the workflow, it can create a *new target*, and *update* an existing target.

Incrementality

ETL does not support any type of *incrementality*.

Directionality

The transformation *directionality* in ETL is unidirectional.

Tracing

The *tracing* support is provided by ETL.

5.4.3. Further Comments

Compared to other technologies, it took less time to get implemented. This hybrid approach of declarative and imperative implementations, such as, ATL makes the implementation more flexible. The model querying is very similar to OCL.

The approach of Epsilon of having one specific language for each task, such as EML for merging models and ETL for transforming models, creates a clear distinction of implementations in the source code. By using a workflow, it is possible to create an execution path that calls the source code from each language, so that it is possible to integrate different Epsilon transformation languages in the transformation process. In the case study implementation, only the ETL language was used. This clear distinction of programming languages divided by tasks, can make the source code and a transformation project better organized by tasks, but in the other hand, it increases the workflow complexity.

One tool that in our point of view is missing in Epsilon is a debugger. A workaround would be the use of the `println()` function to write one specific text in the console. A debugger in a transformation process is very good for analyzing queries, checking variables and using breakpoints.

The `post` block available in Epsilon also creates a lot of flexibility, it was possible to implement the entire bad smell detection in the case study in the `post` block. In ATL it was necessary to create another transformation process for implementing the bad smell detection block, since it does not provide this type of feature.

Epsilon EOL uses the `operation` keyword for declaring operations. In UML, the `operation` keyword is used as an attribute from the `CallOperationAction`. It was not possible to execute the transformation using the `operation` keyword in the query, a workaround is to use its getter to access the field's value such as is shown in Listing 5.4.

```
1 return aBehaviorDiagramsList−>select(e |
2         e.isTypeOf(Source!Activity))−>collect(f | f.node)−>flatten()−>
3           select(g | g.isTypeOf(Source!CallOperationAction))−>select(h |
4             h.getOperation() = self)−>size()−>toString()−>asReal();
```

Listing 5.4: EOL `operation` keyword issue

Another issue found, as in the case of ATL, there is a difference between using `0` and `0.0`. `Integer` values should always be compared to `Integer` values, and `Real` values should be compared with `Real` number, such as `0.0`. Mixing them does not generate an exception in the execution, but the comparison is not resolved correctly.

5.4.4. Conclusions

The conclusions described below are based on the evaluation schema, transformation language classification, and on the case study implementation. Each characteristic is evaluated individually.

Efficiency

Considering the *efficiency* provided by the ETL **tooling**, the time needed for executing the case study implementation was a few milliseconds faster than the slowest transformation

language execution. It was more than two times slower than the fastest transformation execution.

Portability

Considering the *portability* of its **tooling**, it is not available in the standard Eclipse Modeling Tool IDE installation, so that it must be installed separately. This is one of the facts that contributed to the extra time needed creating a new project, configuring it, and running the template project without errors, since the **tooling** had to be installed first. But no problems or difficulties in the **tooling** installation were found. Concerning the *portability* of the **transformation projects**, the few **transformation projects** available found, were installed in the Eclipse IDE without problems.

Usability

The Epsilon ETL *usability* characteristic, concerning its **tooling** and **language features**, contributed to the fastest case study to be implemented considering the other three implementations. **language features** such as the use of `operations` in conjunction with rules makes the implementation more flexible. These `operations` are functions that supports parameters and may return a value or object, similar to other programming languages which makes ETL easier to understand and learn. The use of the post block for implementing the bad smell detection after the metrics calculation also facilitates the implementation using ETL. In other implementations it was necessary to create another transformation implementation for detecting the bad smells. Concerning its **tooling**, the editor does not provide the live detection of errors in the implementation, and the autocompletion feature that enhances *usability* of the **transformation projects** implementations and the **language features**. The *tracing* feature provided by its **tooling** enhances the understandability of a M2M **transformation projects** executions by generating some kind of footprints of the **transformation project** execution that can be analyzed by the developer to better understand the transformation execution.

Analyzing the publicly available third party **transformation projects** found, it was possible to find simple ETL examples, to more complex ones, including projects that not only uses ETL, but also other Epsilon languages that are used combined, providing good examples for learning and understanding the ETL **tooling**, **language features** and the **transformation projects** themselves. Considering the technology that can be used in conjunction, Epsilon provides a set of languages specialized for specific tasks, all of them are built on top of EOL, including ETL. Since the base language is the same, the learning process of the **language features** is much faster, once one of the other languages is known. Well known technologies such as Ant, EMF, and Java can also be used in conjunction, and so enhancing

the **tooling** *usability*, since it avoids the need of learning technology specific **tooling** that is only used in this technology.

Considering the learning material found in 20 minutes research such as books, articles, and tutorials it was possible to find one up-to-date eBook about Epsilon, containing all specifications about all languages. It was also possible to find a few articles related to the ETL language. The learning material in the official website is well structured, and contains good material about *ETL*, including the eBook mentioned above [118] containing the entire Epsilon specification, screencasts, and an embedded application for executing EOL instructions. By analyzing the Epsilon forum (Figure 5.6) it is possible to observe that there are no unanswered topics. The only unanswered topics found were announcement topics from the Epsilon developer. The learning material, and the good community support provides a good source for learning and understanding of its **tooling**, **language features**, and **tools implementations**.

Maintainability

Considering the *ETL* **tooling**, it does not provide a debugger tool for supporting code *maintenance*, that is useful tool for analyzing bugs, and evaluating long queries. Such a tool would enhance the *maintainability* of **transformation projects** by providing the developer with the possibility of identifying erroneous executions of the transformation processes using debugger features, such as breakpoints and evaluating variables in one specific part of the transformation process. An editor function for refactoring the source code, that would contribute to a good *maintainability* of the **transformation projects**, is also not available. Its **tooling** provides the *tracing* feature for tracing M2M transformation processes, an useful feature used for generating the footprints of the transformation process, and so enhancing the maintainability of **transformation projects**.

The lower amount of implemented lines in the case study could indicate that the **transformation projects** are easier to *maintain* due to **transformation language** features that enhance the **transformation projects** *maintainability*, such as *inheritance* of rules, the distribution of rules in different files are provided in ETL, and the use of `operation`. In Table 5.3 all releases in the last two years are listed. It is possible to observe that the last release contains bug fixing and by analyzing the release documentation, it is also possible to observe that not only bug fixing has been done, but enhancements were implemented, that is a good indicative that **tooling implementation** is been *maintained*. Enhancements of the **language features** can also be proposed in the bug tracking system, enhancing the *maintainability* of the **tooling implementation**.

Analyzing Figure 5.4 it is possible to analyze the *maintainability* of its **tooling implementation**, and **language features**. A steep growth of bugs in 2009 is observed, that could indicate quality issues due to its *maintainability*. The amount of enhancements entries is larger than in other transformation language observing it proportionally. In Figure 5.5 are

the bug entries grouped by resolution. It is possible to observe that the amount of fixed bugs raised considerable, this could indicate that more developers are working in the **tooling implementation**.

Functionality

Technologies that can be used in conjunction to ETL, that enhances the **tooling** *functionalities* are the integration of Apache Ant for automating tasks, Epsilon transformation languages that can be combined and used together, providing *transformation projects* with more functionalities. ETL can be executed in Java standalone applications, and can create and call methods from Java objects, features that enhance the *functionalities* of the **transformation projects**.

The **tooling** provided contains an editor, and a supplementary set of tools such as a front-end for GMF, and an EMF tree-based editor for displaying models. The editor source code autocompletion function could not find elements from the metamodels in the autocompletion. This feature is very useful especially in accessing complex metamodels such as the UML metamodel. It also does not provide live error detection such as it is available in other editors.

The *functionalities* provided by the **language features** makes it suitable for detecting bad smells, metric calculations, and also for performing refactoring. It is a hybrid transformation language that supports imperative and declarative implementations. It provides a good *phasing* support compared to ATL, by also providing a block that is called implicitly by the transformation process at the end of the transformation, called `post` block, that is only called if the block has been declared. It also supports more than one source and target model, *exogenous* transformations, and *intermediate structures* support. The *in-place* transformation, that is a good feature in the refactoring processes, is not supported in ETL. An Epsilon language that supports *in-place* transformations, and can be used in refactoring processes is EWL.

Continuity

ETL provides a few publicly available third party **transformation projects** for evaluating the transformation language. These projects are important for developers that are evaluating transformation approaches to analyze its features before adopting the transformation language in projects, so that they are a relevant aspect in the *continuity* of the transformation language. No company is officially supporting the Epsilon project, only indirectly by individual developers in companies that are using Epsilon. A large amount of companies that supports a technology is a good indicative of *continuity*, since it can indicate that these companies are somehow using or providing professional support for the analyzed technology.

The project is coordinated by a small group of people. The enhancements and bug fixing are discussed by a small group of people, official committers are currently two, and two new committers are expected in a near future. Analyzing the group that coordinates the project is an important factor in evaluating a technology, since strategic decisions that imply directly in the *continuity* of the project is taken by this group. Considering the material found during the 20 minutes research time, it was possible to observe a considerable amount of articles found that were written by a specific group of authors.

The release frequency is high, the last release contains not only bug fixing, but there are also enhancements, which shows that not only bug fixing is been done, but also new functionalities are implemented, a good indicative of *continuity*. Observing Figure 5.6 there is a growth in the amount of messages posted, this indicates that the community of Epsilon users is growing, another good indicative of *continuity*. The fact that all messages are answered is also a good indicative for new possible developers, since it guarantees some level of feedback from the community regarding questions.

5.5. Operational QVT

QVT is a specification, therefore, there are many different implementations from QVT. Only the Eclipse QVT implementation in considered in this evaluation as tooling. The topics related to the research of the amount of books, articles, and tutorials in 20 minutes listed in Table 5.1 are evaluated based in the QVT specification, and not based on the Eclipse Operational QVT implementation.

5.5.1. Schema Evaluation

The **time needed to execute the compiled implementation** in Operational QVT is 1,870 seconds[8]. **Configuring and compiling a template project without errors** was fast, only 30 minutes were needed to reference all metamodels, models, define an output file, and implement one simple entry point function for the transformation. Since the QVT plugin is available in the standard Eclipse Modeling Tool installation, it was not necessary to install any other plugin. There is also a GUI for configuring the transformation process, so that it is possible to set the input and output models, the metamodels, and set the tracing configurations. The only point that took time to figure out was to register the BSDMC_MM. To register this metamodel, it is necessary to open the projects configuration, enter in the *QVT Settings* menu, then select the *Metamodel Mappings* and then register the metamodel.

The **number of the case study implementation lines**, not considering the source code documentation is 527 lines. The **time needed for the implementation of the entire case study** was approximately 40 hours.

[8]This value has been calculated by taking the average time of 10 consecutive executions.

The QVT **tool support** in Eclipse is provided by an editor that supports code completion, syntax highlighting, outline view, and error detection. Another tool is a GUI for configuring the transformation process. No free debugger is available for Eclipse. According to the official forum [25], there is one debugger shipped as part of the commercial tool *Borland Together* [58], and an open source debugger implementation is in progress. Another open source QVT plugin for Eclipse is *SmartQVT*[57]. This plugin is not shipped with the standard Eclipse Modeling Tool installation, and its last release is more then one year old[9].

The **amount of public transformation projects available** for the Operational QVT implementation from Eclipse are two. Both are available in the standard Eclipse Modeling Tool installation. Since QVT is an OMG standard, other implementation examples that uses the models supported by the Eclipse QVT implementation should, in theory, also run in the QVT Eclipse implementation. None **metric calculation and bad smell detection public projects were found** that uses QVT.

The **different technologies that may be used in conjunction** are OMG OCL, Apache Ant for automating the transformation process, standalone Java applications, such as described in [75], and considering the Eclipse implementation an EMF integration. It also provides *Black Boxing* mechanisms for including non-QVT implementations to the transformation process. In the *acknowledgments* section in [128] are listed all companies and institutions that contributed or supported the OMG QVT specification. This way it is not possible to exactly define the **number of manufactures that officially support the use of this technology**. The number of companies and institutions listed is 27, between them are companies such as Borland, Hewlett Packard [35], Sun Microsystems that since 27.01.2010 belong to oracle [53], and Thales, as also institutions such as INRIA, University of Paris VI [67], Kings College London [41], and University of York [65].

The **group responsible for the technology maintenance and coordination** is the OMG. This group was founded in 1989, and according to [128] it is a not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. This group is also responsible for the specification from UML, MOF, XMI, OCL, Common Warehouse Metamodel (CWM), SMM, and others. According to the M2M proposal [27], the QVT implementation in Eclipse is coordinated by Borland.

Considering the **release frequency in two years**, it was possible to find in the official Eclipse Operational QVT website four stable version for download. The table containing the last releases is shown in Table 5.4. It was not possible to find the **last official release containing new features**. There were no references about the new versions enhancements and bug fixing. Observing the projects in Bugzilla, it was possible to find one enhancement before the release of version 2.0.1. But it does not guarantee that this enhancement is available in this version.

[9] checked on 31.01.2010

Table 5.4.: Operational QVT releases in the last two years

Version	Date
2.0.1	01.09.2009
2.0.0	16.06.2009
1.0.1	08.11.2008
1.0.0	11.06.2008

The **Bug tracking system analysis** in Eclipse Bugzilla in the last five years[10], filtering the project M2M and components *QVT_OML-Doc*, *QVT_OML-Engine*, *QVT_OML-UI*, *QVT_OML-Website* are shown in Figures 5.7 and 5.8. No bug entries were found from the period of 2005 to 2006 in the Eclipse Bugzilla system.

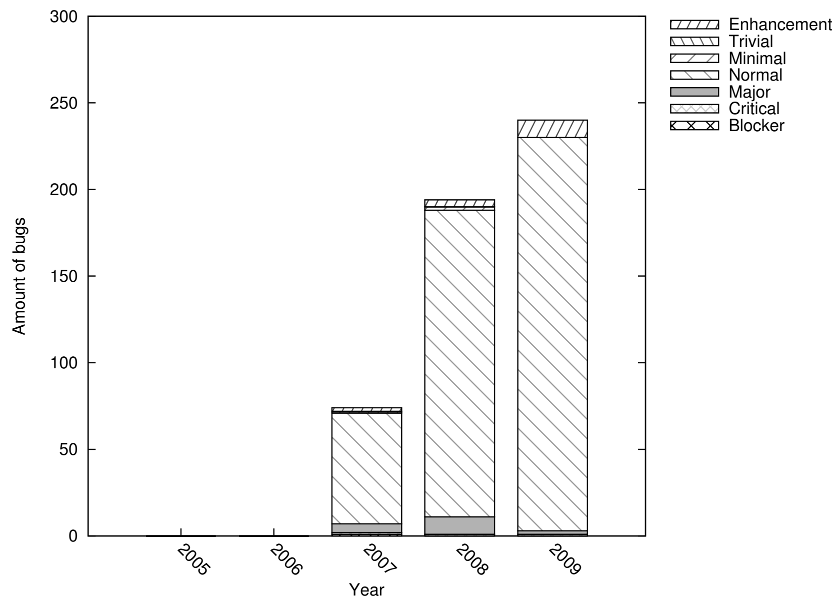

Figure 5.7.: Eclipse Bugzilla QVT projects entries grouped by severity.

[10] 2009 until 08.12.2009.

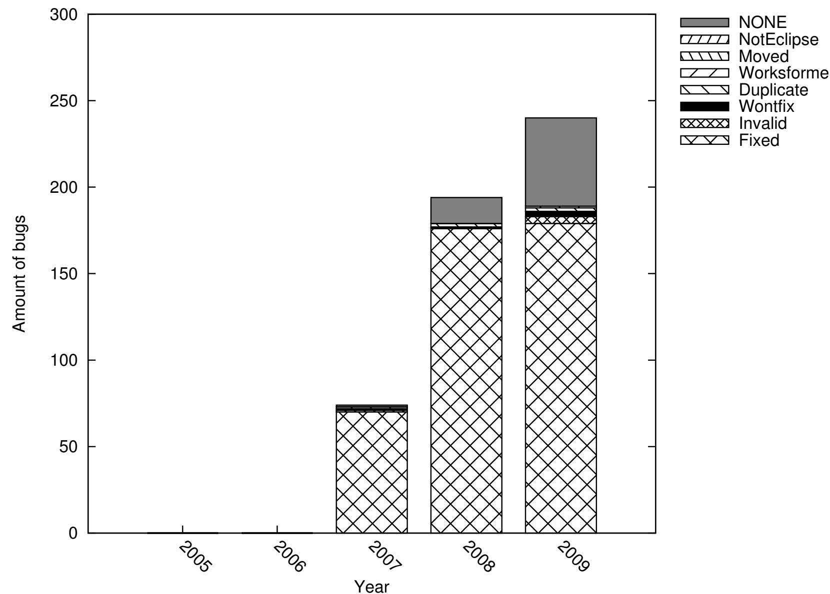

Figure 5.8.: Eclipse Bugzilla QVT projects entries grouped by resolution.

The **amount of books not older then 2 years found in 20 minutes research** is 5. From this set of books, two are about QVT [123, 124], and 3 contains references or chapters related to QVT. The **amount of articles not older then 2 years found in 20 minutes research** is 20. Interesting articles found, in our point of view, are [119, 121]. The **amount of tutorials found not older then 2 years in 20 minutes in research** are two, but none of them are specific for the Eclipse Operational QVT implementation.

The **learning material and information about the technology provided by the official website**, considering the Eclipse Operational QVT project, is very scarce, it has a wiki that have very few topics, a forum, and the only document available is [94]. Since QVT is a specification, an other QVT implementations are available, it is possible to find a large amount materials in other websites.

Figure 5.9 shows the **quantity of topics and replies in the official form from the last five years**[11], that are based on the data collected from [25], and filtered by messages related to QVT. The number of messages found in 2009 is 467.

[11] 2009 until 02.12.2009.

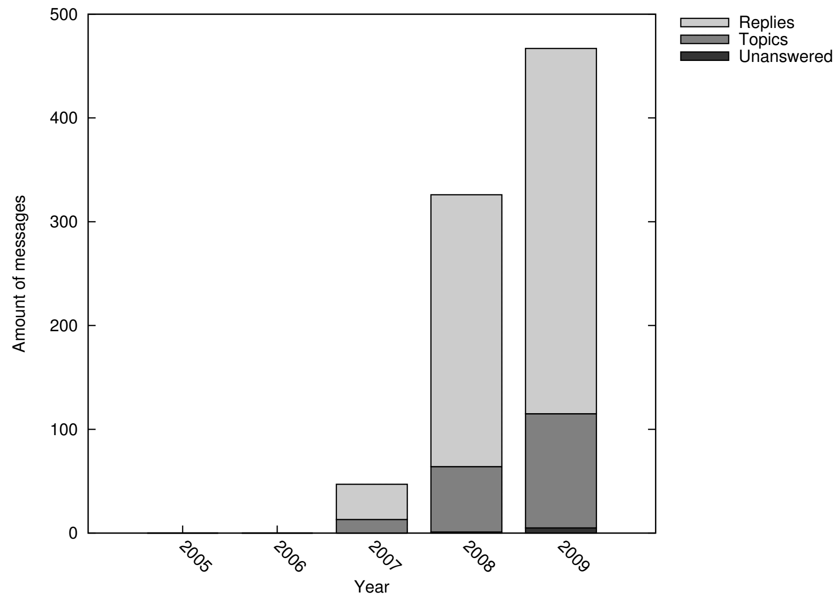

Figure 5.9.: QVT forum usage.

Considering the **quantity of topics and replies in the official forum from the last five years**, it is possible to observe that the amounts of messages found in the forum are increasing in since 2007. The amount of unanswered topics is also very low. The **quantity of topics and replies in the official forum from the last five years** is 840.

5.5.2. Language Characteristics

The Operational QVT transformation language is evaluated based on [89] is described below.

Transformation Rules

Transformation rules, in Operational QVT, do not provide a clear *syntactic separation*. In general cases a left-right *syntactic separation* can be found in mappings, in which left side in an element from the target model, and the right side is the element from the source model. Such as it is shown in Listing 4.15. *Application controls* is also supported using `when` blocks in mappings, to define guard conditions. Post conditions are supported by using `where` blocks inside mapping operations. *Multidirectionality* is not supported in Operational QVT, so that the transformation is always executed in one direction, differently from QVT Relational and Core, that supports *Multidirectionality*. It also supports *intermediate structures*,

by creating a second model in the M2M transformation for other purposes as tracing. The rules *parameterization* supports *control parameters*, that are values passed as parameter, *high-order rules*, but does not support *generics*, such as element types as parameters.

The *domain languages* supported by Operational QVT are Ecore based metamodels, MOF, UML2 metamodels. The transformation can be *endogenous*, or *exogenous*. The domain *static modes in*, *out*, and *in/out* are defined in the transformation header. The *dynamic mode restriction* is considered partial, since it is not possible to change the *static mode* from domains during the transformation, but it is possible to set the *static modes* of values passed as parameters in mappings, that could be part of the target or source model. The *body* supports *variables*, *patters*, and its executable *logic* is based on a *imperative* language, that supports rules *recursion*, and the *value specification* is *imperative assignment*, and the *element creation* is *explicit*. Operational QVT is considered *syntactically typed*.

Rule Application Control

The *rule location determination* is *deterministic*, since for each transformation process exists only one entry point, and the order of rules called is defined in the user implementation.

The *rule scheduling form* is *explicit* and *internal*, since the rule scheduling is implemented by the user, within the transformation implementation, and not implemented in a different file, or diagram. The *rule iteration* supports mechanisms of *recursion*. Since the rules calling order is defined by the user in the implementation, the user can define exactly what transformation is going to be executed in one specific phase of the transformation process, so that *phasing* is supported.

Rule Organization

The *rule organization* concerning the *modularity mechanisms*, is supported in Operational QVT in the form of libraries, and in the form of modules that can be imported to a transformation. The *reuse mechanisms* consist of the import of libraries by using the facilities *access* and *extension*. Importing modules by using the import keyword for accessing for example, other mappings. In mapping operations, Operational QVT supports mapping *inheritance*, *merge*, and *disjunction*. Its *organizational structure* is independent, but in many cases it is *source-driven*.

Source-Target Relationship

The *Source-Target relationship* in Operational QVT can create *new target*, it can perform *in-place* transformations, and *update* in *existing targets*.

Incrementality

Incrementality is not available in Operational QVT, since the target model is always overwritten.

Directionality

The transformation *directionality* is unidirectional, since Operational QVT is an imperative language.

Tracing

The *tracing* support is provided by QVT specification, and available in the Operational QVT implementation.

5.5.3. Further Comments

The inheritance implementation in QVT is more flexible than compared to the other transformation languages evaluated. Listing 5.5 shows in inheritance implementation in our case study.

```
1  mapping Source::Class::AbstractCalculateMetricIncludeName() :
2                                  BSDMC_MM::MetricResult {
3   targetObject := self.name;
4  }
5
6  mapping Source::Class::AbstractCalculateMetric(): BSDMC_MM::MetricResult
7     merges Source::Class::AbstractCalculateMetricIncludeName {
8     init {
9        log('Executing AbstractCalculateMetric() INIT Block',null,0) when
10          self.name = 'X990';
11    }
12    log('Executing AbstractCalculateMetric() BODY Block',null,0) when
13       self.name = 'X990';
14    targetObjectType := self.metaClassName();
15    parentObject := self.package.getQualifiedName();
16    parentObjectType := self.package.name;
17    end {
18       log('Executing AbstractCalculateMetric() END Block',null,0) when
19           self.name = 'X990';
20       }
21 }
22
23 mapping Source::Class::CalculateMetricCBO(): BSDMC_MM::MetricResult
24    inherits Source::Class::AbstractCalculateMetric {
25    init {
26       log('Executing CalculateMetricCBO() INIT Block',null,0) when
27          self.name = 'X990';
28       }
29
```

```
30      log('Executing CalculateMetricCBO() BODY Bock',null,0) when
31         self.name = 'X990';
32    id := 1;
33    text := 'Calculate Coupling Between Object Classes (CBO)';
34    type := MetricsTypes::mtCouplingBetweenObjects;
35    self.GetClassesRelatedInRelationships()−>xcollect(f |
36             result.metricResultItem += result.map CreateMetricItem(f));
37    value := result.metricResultItem−>size();
38    end {
39       log('Executing CalculateMetricCBO() END Block',null,0) when
40          self.name = 'X990';
41       }
42 }
```

Listing 5.5: QVT Inheritance example

Interesting to observe is the order that the `mapping` blocks are executed. Listing 5.6 was extracted from the Eclipse *console* after the execution of the case study QVT implementation and it shows the blocks calling order in the mapping. The first block to be executed is the `init` block from the `CalculateMetricCBO`, the next blocks to be executed are the blocks from the *superclass* `AbstractCalculateMetric`. And at the end are executed the blocks `body`, and `end` from the `CalculateMetricCBO` mapping. It is also possible to observe in the listing above the merging of the `AbstractCalculateMetric` and `AbstractCalculateMetricIncludeName` `mapping` in line 7.

```
1 Level 0 − Executing CalculateMetricCBO() INIT Block
2 Level 0 − Executing AbstractCalculateMetric() INIT Block
3 Level 0 − Executing AbstractCalculateMetric() BODY Block
4 Level 0 − Executing AbstractCalculateMetric() END Block
5 Level 0 − Executing CalculateMetricCBO() BODY Bock
6 Level 0 − Executing CalculateMetricCBO() END Block
```

Listing 5.6: QVT inherited class calling order

5.5.4. Conclusions

The conclusions described below are based on the evaluation schema, transformation language classification, and on the case study implementation. Each characteristic is evaluated individually.

Efficiency

After implementing the case study, and analyzing the language based on the evaluation schema, we can observe that considering the **tooling** *efficiency* characteristic, it was the second fastest implementation in the source code execution. Compared to the fastest execution it was more than twice as slow as the fastest execution, but compared to the other two languages, it was over two times faster. This good performance is very important in

detecting bad smells, calculating metrics, performing refactoring in larger models, and in the execution of complex transformations in large models.

Portability

All **tooling** necessary for creating and configuring a new project was available in the Eclipse Modeling Tools IDE standard installation, so that it was not necessary to install any plugin, or library to configure the case study template project. Considering the **transformation projects** *portability*, creating a new project, configuring it, and executing it without errors was fast. Installing the two transformation projects in the IDE where a straightforward process. This QVT implementation is integrated in to Eclipse, but QVT is an OMG specification, so that OML source code implemented in another QVT tool should in theory, be *portable* to the Operational QVT **tooling**. The two **transformation projects** available are shipped with in the Eclipse Modeling Tools IDE, installing the two **transformation projects** was a straightforward process.

Usability

Considering the *usability* of its **tooling**, and **language features**, Operational QVT was the second fastest implementation, especially because of the **tooling** provided. Good features provided by its **tooling**, and **language features** are the code autocompletion including the metamodel structure, and the live error detection in the editor, two features that enhances the understanding, and learning of the **language features**, and the **transformation projects** during the implementation time. The *tracing* feature provided by the **language feature** and its **tooling** enhances the understandability of a M2M **transformation project** execution.

The two publicly available third party **transformation projects** are very well documented projects, one containing a simple M2M transformation, and the other is an example of defining a simple black-box library in Java. This two transformation projects enhances the learnability and the understandability of the **language features**, and **transformation project**. Considering the different technologies that can be used in conjunction, its integration to EMF contributes to the user's understandability, and learnability of the Operational QVT implementation **tooling**, once the user is familiarized to EMF. Apache Ant, and Java are technologies that are not QVT specific, and that are used by the community for other tasks not related to QVT, this enhances the understanding and learning its **tooling**. A factor that also contributes to a better understanding and learning of its **language features** is the support of the OCL standard.

Considering the research of learning material in 20 minutes, two books were found, in which it was the only transformation language that had books written especially for the transformation language. It has the highest amount of literature found during the research period. It provides a good source of information for learning and understanding the **tool-**

ing related to the Operational QVT implementation, **language features**, and the Operational QVT *tools implementations*. The material found in the Operational QVT implementation in Eclipse is very scarce compared to the other transformation materials. But since QVT it is a specification, other material about the QVT OML can be found in other QVT implementations websites, providing a good source of information of the *usability* of its **language specification**. The forum activity (Figure 5.9) shows that it is an active forum, and has a small amount of unanswered topics, providing a good source of information of its **tooling**, **tools implementations**, and **language features**.

Maintainability

Considering its **tooling**, a debugger, and a refactoring tools that contributes to a better *maintainability* of the **transformation projects** are not available in Operational QVT. A debugger is a very important tool in analyzing bugs, especially in analyzing OCL queries in large models. The refactoring tool is an important tool that can be used in M2M **transformation projects** implementations. A good project documentation, containing information about each release is a good indicative of *maintainability*. Considering the *maintainability* of its **tooling**, and the **tools implementation**, the releases in the last two years in Table 5.4, it is possible to observe that the time between releases it is relatively constant, so that it is being maintained with bug fixing. It was not possible to find a list containing which bugs where fixed, so that it is not possible to identify the last release containing new features.

Analyzing Figure 5.8, it is possible to identify that the amount of entries in the bug tracking system is not increasing so rapidly as in the other evaluated languages, it could be an indicative of a good *maintainability* of the **tooling implementation**. It is also possible to observe that the amount of bugs fixed in the last two years is relatively constant, and the amount of open bugs increased. It could mean that more developers in the project are needed.

Language features that enhance the *maintainability* of the **transformation projects** are *inheritance*, and *merge* of `mappings` as also from entire transformations, it is possible to divide the implementation in different modules. The amount of the implementation source code lines is 15% smaller than the larger implementation. This result can be due to its *reuse mechanisms*, and also due its compactness in writing OCL queries. The *tracing* feature is specified in the QVT specification, and also implemented in the Operational QVT **tooling**, providing a good support for analyzing and detecting bugs in the M2M transformation processes.

Functionality

Considering the *functionality* characteristic related to technologies that can be used with Operational QVT **tooling** in the Eclipse platform, Ant can be used for automating **trans-**

formation projects task. Specified in the QVT specification, and available in the Operational QVT **tooling**, *Black-Box* functionalities provides the **transformation projects** with the possibility of calling other programming language code during the transformation process execution. The *functionalities* provided by the Operational QVT **tooling**, is a GUI for configuring the **transformation projects**, editor *functionalities* such as code completion including the UML metamodel, and the live error detection.

Considering the **language features**, Operational QVT is an imperative language, its *rule application control* is implemented by the user, so that the *rules scheduling* is defined by the user in the implementation. Since the rules execution order is defined by the user in the implementation, *phasing* is fully supported. This feature can be used in bad smell detection process, so that it is divided in two phases, one for detecting metrics, and the other for detecting the bad smells based on the metrics result, such as implemented in the case study. It also supports *intermediate structures*, that can be used in implementations for performing refactoring in a *in-place* transformation for example, in which temporary data is stored in the target model during the transformation process, but it is not saved in the result model. Such as in the other transformation languages it also supports *exogenous* transformations.

Continuity

The two publicly available third party **transformation projects** are a good source of information for the evaluation in the adoption of QVT in a project, especially showing the flexibility of the Operational QVT implementation in Eclipse. There is one simple transformation project , and another one providing *Black-Box* functionalities.

Quantitatively, it also has the largest amount of companies and institutions that support the specification. Qualitatively, many of the listed companies are in the IT sector. This large amount of companies that supports the specification is a good indicative of the technology *continuity*. The QVT specification is coordinated by OMG, that is also responsible for the coordination of other important specifications related to MDA, as for MDA itself. The Operational QVT component in Eclipse is coordinated by Borland, that is a company with tradition in the IT market, another good indicative of the Operational QVT *continuity*, as also a relatively constant release frequency.

Considering the bug tracking system analysis, it is possible to observe an increase of bugs in the last year, that could indicate that the community that uses Operational QVT is growing. Analyzing Figure 5.9, there is a growth of messages posted in the forum, and this could be an indicative that the community of Operational QVT users is growing. The amount of unanswered messages is very low compared to other transformation languages, it is also a good indicative for possible new developers, since it guarantees some level of feedback from the community regarding questions.

5.6. Xtend

The Xtend transformation language was formerly available in the oAW project, but in its new version it is part of the Eclipse Xpand project. In the evaluation schema are listed topics that evaluate the Xtend language in a broader way, by evaluating characteristics related to its project. These topics are those related to project coordination, bug tracking system analysis, forum usage, tool support, and the material available in the official site. All other topics are evaluated exclusively taking in consideration only the Xtend language. In section 5.6.2 is only the Xtend language is evaluated.

5.6.1. Schema Evaluation

The **time needed to execute the compiled implementation** was 4,110 seconds[12]. **Configuring and compiling a template project without errors** took 210 minutes to reference all metamodels, models, define an output file, and implement one simple entry point for the transformation.

The **number of the case study implementation lines**, including the workflow, and not considering the source code documentation is 582 lines. The **time needed for the implementation of the case study** was approximately 60 hours.

The **tool support** is composed by an editor that is integrated to the Eclipse IDE providing the user with code completion, syntax highlighting, error detection, an outline view, GUI for configuring the projects workflow, workflow syntax highlighting, and a debugger. The refactorings available in Xtend are *rename extension*, *extract extension*, and *move extension*.

The **amount of public transformation projects available** in the official website is none. There are no Xtend transformation projects available in its official website. The **metric calculation and bad smell detection public projects found** was also none.

The **different technologies that can be used in conjunction** with Xtend are Xpand and Check languages, the oAW workflow for automating the transformation process, Java by calling Java implementations during the transformation process, XWeave [102] is a model weaver that supports weaving models and metamodels, and the EMF framework integrated technologies that can be used in conjunction.

The **number of manufactures that officially support the use of this technology** are five, considering the companies that provides training and support for the Xpand project are listed in [50]. It was also possible to find a list of 8 institutions and companies that used the oAW project, including Itemis [39], and Borland.

The **group responsible for the technology maintenance and coordination** is the oAW group. As it is described in [49], it is composed by a group of people, and not companies. But it is expected that most team members act on behalf of companies, and spend at least

[12]This value has been calculated by taking the average time of 10 consecutive executions.

some of the work time working on oAW. The team is between other tasks responsible for voting on the inclusion of additional core components and addons, responsible for building, maintaining, and running the oAW infrastructure such as Concurrent Versions System (CVS), bug tracking system, build, and websites.

The **release frequency in the last two years** are shown in the table Table 5.5 below. The oAW project was until version 4.3.1 an oAW project. By integrating it to the Eclipse modeling project, it was necessary to adapt the version to the Eclipse version structure. In Eclipse, every incubated project must start with version 0. As in the Xpand project case.

Table 5.5.: oAW and Xpand releases in the last two years

Version	Date
0.7.2	12.08.2009
0.7.1	17.07.2009
0.7.0	16.06.2009
4.3.1	22.12.2008
4.3.0	05.05.2008

According to the Eclipse Xpand release notes, the **last official release containing new features** is Xpand version 0.7.1.

The **Bug tracking system analysis** in Eclipse Bugzilla in the last five years[13], filtering the project M2T and component Xpand are shown in Figures 5.7 and 5.8. No bug entries were found from the period of 2005 to 2006 in the Eclipse Bugzilla system.

[13] 2009 until 08.12.2009.

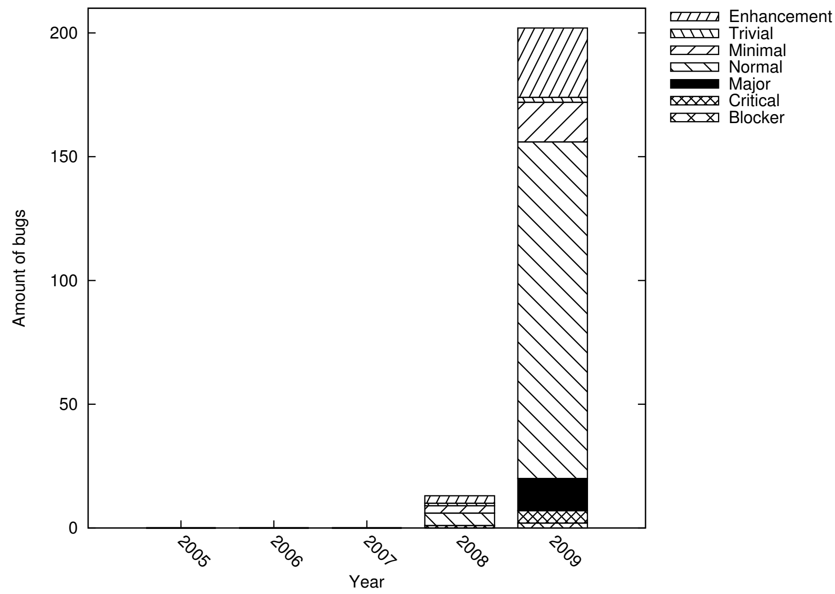

Figure 5.10.: Eclipse Bugzilla Xpand projects entries grouped by severity.

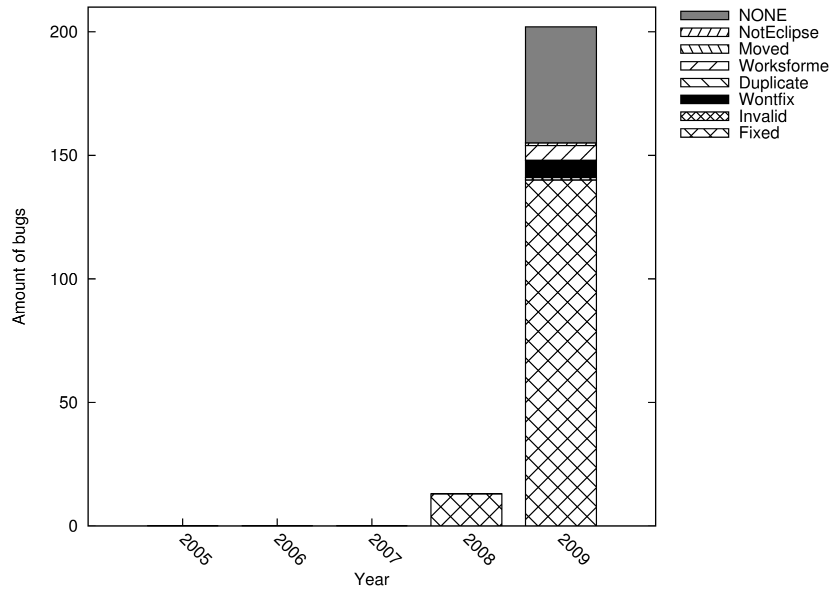

Figure 5.11.: Eclipse Bugzilla Xpand projects entries grouped by resolution.

The **amount of books not older then 2 years found in a 20 minutes research** was one. The book [103] is not specifically about oAW, Xtend, or Xpand, but contains good information about the use of Xtend. It was also possible to find 2 books older than two years in this research period. The books found are [135, 143].

The **amount of articles not older then 2 years found in a 20 minutes research** was nine. A few of the articles found are [90, 96, 113]. The **amount of tutorials not older then 2 years found on a 20 minutes research** is one. It was possible to find a tutorial about using oAW in [46]. It was also possible to find a series of screencasts in [51].

Concerning the **learning material and information about the technology provided by the official website**, it is composed by a tutorial [46], access to the source code, Bugzilla access, developer information, a forum available in [26], download page, an empty FAQ, a detailed plan about the Xpand 0.8.0 release, an empty wiki, list of committers and contributors, and newsgroup. The grand majority of information about Xpand can still be found in [47]. This webpage is the old official site from Xpand. There it is possible to find among others a large amount of documentation, screencasts, a forum related to the oAW Version 4, and a list of companies that provides training and consulting.

There are two official forums, the forum M2T from Eclipse [26], and one forum based on the oAW 4 version available in [48]. The Eclipse M2T forum is not exclusive for the Xpand project, so that the messages related to the Xpand project were filtered out. The messages from the oAW 4 forum, consists of all messages related to the oAW project in the older forum. Figure 5.12 shows the **quantity of topics and replies in the official form from the last five years**[14]. The **quantity of topics and replies in the official forum from the last five years** is 11489.

[14] 2009 until 02.12.2009.

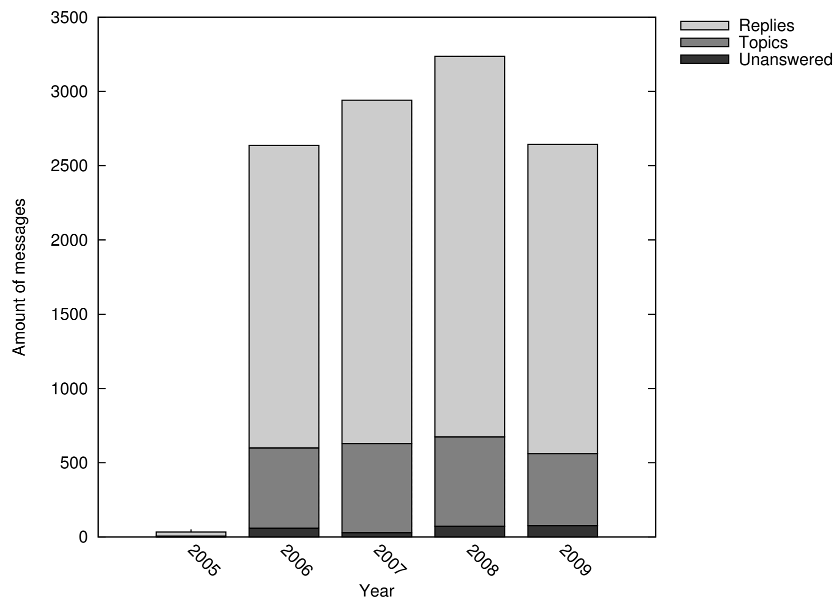

Figure 5.12.: Xtend forums usage.

5.6.2. Language Characteristics

The Xtend transformation language is evaluated based on [89] is described below.

Transformation Rules

Transformation rules, do not provide a clear *syntactic separation*. But in general, in the case of a function, it is a left-right *syntactic separation*. A function creates a new instance using the keyword `create`, the values passed as parameters could be considered as the left side of the transformation, and the `return` object created, the right side of the transformation. *Application controls* are supported by ternary operations, or by Java implementations. Xtend do not support *multidirectionality*, so that for executing one transformation in the other direction, one new implementation is needed. It also supports *intermediate structures*, by creating a second model in the M2M transformation for other purposes such as tracing. Considering the rules *parameterizations*, Xtend supports *control parameters*, that are values that are passed as parameter, *generics* that allows data or element types being passed as parameter, but does not support *high-order rules*, that are other rules that are passed as parameters. According to [147], Xtend does not support *Reflection*, and *Aspects*.

The *domain languages* supported by Xtend are UML2, ECore, IBM Rational Software Architect / Modeler, XML Schema (XSD), oAW-Classic Metamodel, and JavaBeans meta-

model. The transformation can be *endogenous*, or *exogenous*. The *static modes*, *in*, *out*, and *in/out* are defined in the workflow. *Dynamic mode restriction* is not supported in same Xtend transformation, since the models *static modes* are defined in the workflow. The *body* supports *variables*, *patters*, and its executable *logic* is based on a *imperative* language, that supports rules *recursion*, and the *value specification* is *imperative assignment*, and the *element creation* is *explicit*. Xtend is considered *syntactically typed*.

Rule Application Control

The *Rule location determination* is *deterministic*, since for each transformation process exists only one entry point, and the order of rules called is defined in the user implementation.

The *rule scheduling form* in Xtend is *explicit* and *internal*, since the rule scheduling is implemented by the user, within the transformation implementation, and not implemented in a different file, or diagram. The *rule iteration* support mechanisms of *recursion*. Since the rules calling order is defined by the user in the implementation, the user can define exactly what transformation is going to be executed in one specific phase of the transformation process, so that *phasing* is supported.

Rule Organization

The *rule organization* concerning the *modularity mechanisms*, it is possible to organize the rules in *modules*, known as *Extensions* in Xtend. According to [147], *reuse mechanism* are provided in Xtend by using Java extensions. Its *organizational structure* is independent, but in many cases it is *source-driven*.

Source-Target Relationship

The *Source-Target relationship* in Xtend can create *new target*, it can perform *in-place* transformations in *existing targets*.

Incrementality

Incrementality is not supported in M2M transformations in Xtend, since the target model is always overwritten.

Directionality

The transformation *directionality* is unidirectional, since it is an imperative language.

Tracing

The *tracing* support is provided by Xtend, but it must be explicitly added to the transformation code according to [73].

5.6.3. Further Comments

The first case study implementation was in Xtend. During this implementation it was also necessary to create the logic for querying the UML model. The debugger functionality was very helpful for implementing the queries, especially the more complex ones.

All implementations use the same logic for calculating the bad smells. Since this was the first transformation language implemented, the time needed to understand the UML metamodel for accessing and calculating the metrics were subtracted from the time needed for implementing the case study using this transformation language.

Such as commented in section 4.5, for performance reasons, Xtend does not execute one function again if the same values are passed as parameters, one workaround is to pass one extra parameter that is always different, such as a counter variable.

5.6.4. Conclusions

The conclusions described below are based on the evaluation schema, transformation language classification, and on the case study implementation. Each characteristic is evaluated individually.

Efficiency

Considering the *efficiency* of its **tooling**, the time needed for executing the case study implementation was the highest one. The fastest implementation was over five times faster than the Xtend execution. This execution time is similar to the third fastest implementation.

Portability

The **tooling** provided is available in the Eclipse Modeling Tools IDE standard installation, so that it was not necessary to install any plugin, or library. Analyzing the *portability* of the **transformation projects**, the creation of a new project, configuring it, and executing it without errors, took more time compared to other implementations. It was not possible to find **transformation projects** in the official website. Xtend **transformation projects** implemented in the oAW 4 must be migrated to the oAW 5 **tooling** as described in [74].

Usability

Concerning the *usability* characteristic of its **tooling**, and **language features**, the time needed for implementing the case study, compared to other transformation languages. The fastest implementation was twice as fast as the Xtend implementation. The **tooling** provides a good source autocompletion, and a good live error detection of the source code, that enhances the learnability and the understandability of the **language features** and **transformation projects**. The oAW workflow file in the case study implementation contributed to a higher time needed to configure a template project, since it took some time to find a documentation describing the differences between oAW 4 and oAW 5 workflow configuration. The **language feature** inherits features from the *expression language*, also used by other oAW languages, which enhances the understandability, and the learnability of Xtend languages, once one of the two other languages are known. It was not possible to find any publicly available third party **transformation project** that could enhance the learnability and understandability of the **tooling**, and the **language features** in its official website. Technologies that can be used in conjunction with Xtend are the oAW languages Xpand, and Check. Its integration with EMF, and its Java support makes the understanding and learning of the **transformation projects**, and the **tooling** faster.

Considering the number of book, articles, and tutorials found in 20 minutes research, it was possible to find a tutorial about oAW and also screencasts, providing a good source of information of its **tooling**, **language features**, and **tools implementation**. The book [103] contains a chapter about the Xpand project, that explains the *expression language*, Xtend, and Xpand languages. The learning material found in the official website is scarce compared to other transformation languages, but a large amount of material can be found in the old oAW website [47], providing a good amount of learning material for its **tooling**, **language features**, and **tools implementation**. Figure 5.12 shows the forums data collected, based on the old oAW forum, and in the new Eclipse Xpand project forum, it is possible to observe that it has the highest amount of messages, and a low rate of unanswered messages, providing a good source of learning material, and enhancing the usability of Xtend.

Maintainability

Its **tooling** enhances the *maintainability* of the **transformation projects** by providing a good debugger, especially in analyzing variables. In analyzing a variable using the debugger, it is possible to evaluate all child elements from the variables listed in the variables view. It is especially useful in analyzing complex queries in large models where it is possible to evaluate all sub elements from the analyzed variable. One nice feature not available, would be a breakpoint function that only stops on defined conditions. This feature is available in other debuggers [21] and it is very useful especially in analyzing large models. Another **tooling** feature that enhances the **transformation projects** *maintainability* is a refactoring

tool. It provides the refactoring functions *rename extension*, *extract extension*, and *move extension*.

The *tracing* feature provided by its **tooling** for evaluating M2M **transformation projects** for detecting bugs and analyzing the behavior of the transformation execution enhances the *maintainability* of **transformation projects**. One disadvantage of the *tracing* provided in Xtend is that it must be added explicitly in the users source code, differently than in other transformation languages where the tracing can be automatically generated. Analyzing the *maintainability* of its **tooling**, and *tools implementation*, considering the releases, the last version containing new features is the version 0.7.1, but not in the current version 0.7.2.

Considering the bug tracking system entries to analyze the *maintainability* of its **tooling implementation**, and **language features** it was only possible to find bug entries from the years 2008 and 2009. Two bug tracking systems where used in this analysis, the Eclipse Xpand project in Eclipse Bugzilla, and the old Bugzilla located in the oAW 4 website. By analyzing the bug tracking systems, it is possible to evaluate the quality of bugs found in its **tooling implementation**, as also enhancements proposed by the community for its **tooling**, and **language features**. Based on the data collected in Figures 5.10, and 5.11, it is possible to observe that there is a bug policy for bugs classification, it is also possible to identify that a large amount of bugs where fixed in 2009, also considering the small amount of bugs entries in 2008.

Functionality

The Xtend **tooling** is one part of the Xpand Eclipse project, so that the transformation language can be used in conjunction with other oAW languages, such as the Xpand for executing M2T transformations, and the Check language for constraint checking. Another **tooling** that enhances the **transformation projects** *functionalities* is the use of an oAW workflow for automatizing **transformation projects**. Java operations can also be called in Xtend implementation, or Xtend can also be integrated to a standalone Java implementation.

The **tooling** enhances the technology *functionalities* by providing a GUI for configuring the **transformation projects** basic settings, as for example setting the workflow file. The code autocompletion also shows the metamodel element, which is very useful especially in building complex queries. Another **tooling** that enhances the *functionalities* are a debugger, and the live error detection editor's function.

Considering the Xtend **language features**, showed that it can also be used for detecting bad smells, calculating metrics, and also refactoring models. It is an imperative language, so that the *rule application control* is implemented by the user. *Phasing* is supported since the rules calling order is defined by the user. *Phasing* was used in our case study, by dividing in the implementation the metric calculation phase, and the bad smell detection phase. It also support diverse *domains*, useful for working with different models, especially in *exogenous* transformations. In can also execute *in-place* transformations, useful in *refactoring* process,

as also *intermediate structures* are supported.

Continuity

The lack of publicly available third party **transformation projects** available in the official website does not contribute to the adoption of Xtend in projects by new developers. Simple examples of transformation projects that explain how transformations using Xtend works, and showing its features are important for new developers that are evaluating different technologies before choosing one to work with. A good amount of transformation projects and learning material can be found in the old official oAW website, but new developers tend to search for this kind of information in the website where the new version is available. Considering the material found in the research, the book [103] contains a chapter about the Xpand project, but analyzing the entire book, it is possible to observe that the majority of M2M transformations are described using QVT, and not Xtend.

Considering the companies that officially supports Xpand, it was possible to find a group of companies that provide training, and support for Xpand, and also eight companies that used oAW were found, including Borland and Itemis. Companies that provide professional support and training contribute to the technology *continuity*, since it is a factor that is considered by companies before adopting one technology to use.

The group responsible for the coordination of the oAW project is based on persons that are professionally active in the field of software development using oAW, guidelines about the team structure, voting system, roles, tasks, so that it is organized in a very structured way. Such clear guidelines, and developers that are professionally active in the field of software development using oAW can contribute to the enhancement of the software *continuity*.

The last release of the Xpand project does not have enhancements, only bug fixing. It was possible to find enhancements in version 0.7.1. Consecutive version releases without new features is not considered a good indicative of *continuity*.

The transformation language with the most active forum community is oAW as it is shown in Figure 5.12. This large amount of messages is an indicative that oAW is used by a large community of developers, and a good indicative of the technology *continuity*.

6. Conclusion

The use of UML models for specifying structural and behavioral characteristics of software is increasing. The use of tools and methods for ensuring software quality in models nowadays is an important part of the software development cycle. Model transformation languages can be used in the UML quality engineering field, in tasks such as metric calculation, bad smell detection, and refactoring. After analyzing the four transformation languages in this thesis, we conclude that all of them are suitable for UML quality engineering in metric calculation, bad smell detection, and refactoring processes.

Analyzing the transformation languages features, ETL is the only transformation language that does not provide in-place transformation, which is a functionality that is commonly used in refactoring processes. All analyzed transformation languages support intermediate structures, which is also a functionality useful in in-place refactoring processes. The support of exogenous transformation is an important feature provided by all transformation languages, which is useful in developing tools that should work with different metamodel types.

Phasing is also an important feature supported by all transformation languages analyzed, which consists of defining one specific set of rules that should be executed in a specific phase of the transformation process. In the two hybrid languages, ATL and ETL, it is possible to explicitly call rules in imperative blocks of the source code. Operational QVT and Xtend are imperative languages, so that the sequence order of the rules called, is defined in the implementation. An important transformation language feature not available in the analyzed languages is the incrementality feature introduced in [89]. This feature consists of detecting a modification in the source or target model, so that a transformation can be executed only in a specific part model, such as the modified model part. It can increase the tooling efficiency, and is useful in tasks such as models synchronization.

Considering the LLOC metric calculated in each case study implementation, it is possible to observe that the QVT implementation has the smaller amount of source code lines. This compactness can enhance the maintainability of code by reducing the amount of source code lines that must be maintained. This result can be due to its compactness in writing OCL queries, and due to its reuse mechanisms such as rule inheritance and merging. This implementation took 15% less source code lines than the largest implementation. This compactness, the simplicity of the Operational QVT language syntax, and the QVT literature found, also contributed to the low implementation time needed in the case study compared to other transformation languages.

The transformation languages features, and tooling in Operational QVT, ATL, ETL, and Xtend are good documented. This good documentation enhances the usability of these transformation languages. Considering the Xtend documentation, mostly is available in the old oAW website, and not in the Eclipse Xpand project website. Another good source of information about the language features, and tooling are the forums, especially the ATL, Xtend, ETL forums. The ATL and Xtend forums has the largest number of messages compared to the other forums. In the ETL forum it was possible to observe in the research that no messages from the community were unanswered, the only not answered messages are those posted by the developers. This feedback provided by the ETL forum is also a good source of information.

The ATL, Operational QVT, and Xtend tooling is integrated in the Eclipse Modeling Tools installation out-of-the-box, while the Epsilon project, that contains ETL must be installed separately. All the transformation languages tooling provide tracing functionalities, an important feature for analyzing transformation executions in large models, such as in refactoring processes in identifying what elements were modified during the M2M transformation process. Operational QVT and the ATL tooling had a good efficiency in executing the case study implementation, compared to the other transformation languages. The performance was almost two times faster than the other two languages evaluated.

The source code editors provided by the transformation languages Operational QVT and Xtend can also display the metamodel structure in the code completion, which is a feature that enhances the usability of the transformation language, especially useful in building large and complex queries. Only the transformation languages Xtend and ATL have a debugger tool, which is important in finding bugs and maintaining the transformation projects. The debugger tool available in Xtend is considered good, since it can not only display the elements from metamodels, but also all sub elements in the form of a tree view in the variables view. A very good feature in debugging and building long queries.

By analyzing the amount of open bugs in the bug tracking system from each of the evaluated transformation languages, Operational QVT has the lowest amount of open bugs related to its tooling and tooling implementation in the last five years. This evaluation considers the amount of bugs proportionally to the total amount of bugs in each transformation language. This is a good indicative that its tooling and tooling implementation is being maintained.

ATL was the transformation language that had the largest number of public available transformation projects. It was also the only transformation language were it was possible to find a transformation project for metric calculation.

The language with a good continuity is Operational QVT. Factors that contributed to this result are the QVT specification by OMG, and the QVT specification support by important companies from the IT sector. Operational QVT is based on the QVT specification, so that there are many tools available in the market that implements the QVT specifications, and in this way contributing in the transformation language continuity.

6.1. Outlook

The results of this thesis can be used as decision point for selecting one transformation language to be used in the development of tools related to UML quality engineering. Several extensions of this thesis are possible, the already implemented code in the case study can be extended for calculating other metrics, bad smells, and for refactoring models. The metamodel BSDMC_MM specified in this thesis provides all necessary structures for executing the above described tasks.

Other transformation languages could be also evaluated based on the analysis presented in this thesis. Such M2M transformation languages are EMF Model Query, Tefkat, the declarative languages of QVT core and relation languages, Kermeta, and ModelMorf. This evaluation can be extended by analyzing not only M2M transformation languages, but also include general purpose languages such as Java to the languages evaluated.

Expanding the evaluation schema presented in this thesis, by creating a second level of topics, specific for each of the subject analyzed, such as tooling and transformation projects, and based on this new level, evaluate how exactly each of the subjects influence other subjects from the evaluated technology considering the ISO 9126 characteristics.

This evaluation schema could also be extended by taking in consideration how each of the transformation languages characteristics based on the study of Czarnecki and Helsen [89], can influence the ISO 9126 characteristics of the analyzed technology.

A larger set of metrics can be added to the evaluation schema. These metrics would be calculated for each of the case study implementations. Based on these metrics it is possible to take further conclusions about characteristics related to the transformation language.

Calculating metrics and detecting bad smells in large models, can be very time consuming. Include to the evaluation schema, topics related to the efficiency of the transformation languages concerning specific tasks in source models from different sizes. Conclusions such as the feasibility in using an evaluated transformation language in specific tasks, taking in consideration specific model sizes, can be taken based on such evaluation.

Another extension of this thesis is the creation of a very detailed methodology for evaluating the continuity classification introduced in this work. This evaluation could be analyzed based on many more factors relevant in predicting the continuity of the evaluated technologies.

List of Tables

5.1. Evaluation Schema 61
5.2. ATL releases in the last two years 64
5.3. Epsilon releases in the last two years 76
5.4. Operational QVT releases in the last two years 87
5.5. oAW and Xpand releases in the last two years 97

List of Figures

2.1. Code Generation Process in MDA 6
2.2. UML Diagrams 9
2.3. UML Metamodeling 10
2.4. ISO 9126 quality model for external and internal quality. 12
2.5. Top Level feature classification 14
2.6. QVT metamodels relationships 23

3.1. Case study complete process 30
3.2. BSDMC_MM Class Diagram 31
3.3. Example model class diagram 34
3.4. Source model *Sequence Diagram* for initializing all objects 35
3.5. *Activity Diagram* from method `X990::initializeVariables` 36
3.6. *Activity Diagram* from method `FuelTank::CalculateAutonomy` 36
3.7. *Activity Diagram* from method `EngineWS67::initializeVariables` 37

5.1. Eclipse Bugzilla ATL projects entries grouped by severity. 65
5.2. Eclipse Bugzilla ATL projects entries grouped by resolution. 65
5.3. ATL forum usage. 66
5.4. Eclipse Bugzilla Epsilon projects entries grouped by severity. 77
5.5. Eclipse Bugzilla Epsilon projects entries grouped by resolution. 77
5.6. Epsilon forum usage. 78
5.7. Eclipse Bugzilla QVT projects entries grouped by severity. 87
5.8. Eclipse Bugzilla QVT projects entries grouped by resolution. 88
5.9. QVT forum usage. 89
5.10. Eclipse Bugzilla Xpand projects entries grouped by severity. 98
5.11. Eclipse Bugzilla Xpand projects entries grouped by resolution. 98
5.12. Xtend forums usage. 100

Listings

2.1. ATL `header` example 18
2.2. ATL `header` syntax 18
2.3. ATL `header` example 18
2.4. ATL `rule` syntax 18
2.5. ATL `rule` example 19
2.6. EOL pre and post condition in operations example 21
2.7. ETL `rule` syntax 21
2.8. ETL `rule` example 22
2.9. QVT *relations* source code example 23
2.10. OML header example 24
2.11. OML example 24
2.12. OML `mapping` general syntax 25
2.13. OML `mapping` example 25
2.14. Xtend example 27
2.15. Workflow of the Xtend case study implementation 28
2.16. OCL example 29
2.17. Another OCL example 29

4.1. ATL transformation initialization 42
4.2. ATL `CalculateMetricInProperties` rule 43
4.3. ATL `CalculateMetricFRiC` rule 44
4.4. ATL `CountRefDBFound helper` 44
4.5. ATL `GetPropertyReferencesInOpaqueActions helper` 45
4.6. ATL `GetPropertyReferencesInCreateObjectAction helper` 46
4.7. ETL transformation initialization 46
4.8. ETL `CalculateMetricInProperties` rule 47
4.9. ETL `CalculateMetricFRiC operation` 48
4.10. ETL `CountRefDBFound operation` 48
4.11. ETL `GetPropertyReferencesInOpaqueActions operation` 49
4.12. ETL `GetPropertyReferencesInCreateObjectAction operation` 49
4.13. Operational QVT transformation initialization 51
4.14. Operational QVT `CalculateMetricInProperties mapping` 52

4.15. Operational QVT `CalculateMetricFRiCs` and `CalculateMetricFRiC` `mappings` . . . 52
4.16. Operational QVT `CountRefDBFound` `query` . . . 53
4.17. Operational QVT `GetPropertyReferencesInOpaqueActions` `query` . . . 53
4.18. Operational QVT `GetPropertyReferencesInCreateObjectAction` `query` . . . 54
4.19. Xtend transformation initialization . . . 55
4.20. Xtend `CalculateMetricInProperties` function . . . 55
4.21. Xtend `CalculateMetricFRiCs` and `CalculateMetricFRiC` functions . . . 56
4.22. Xtend `CountRefDBFound` function . . . 57
4.23. Xtend `GetPropertyReferencesInOpaqueActions` function . . . 58
4.24. Xtend `GetPropertyReferencesInCreateObjectAction` function . . . 58

5.1. ATL comparison issue . . . 69
5.2. ATL query example . . . 69
5.3. QVT query example . . . 69
5.4. EOL `operation` keyword issue . . . 81
5.5. QVT Inheritance example . . . 91
5.6. QVT inherited class calling order . . . 92

A. Acronyms

AGG Attributed Graph Grammar System

AM3 ATLAS MegaModel Management

AMW Atlas Model Weaver

AmmA AtlanMod Model Management Architecture

API Application Programming Interface

ATL ATLAS Transformation Language

AtoM3 A Tool for Multi-formalism Meta-Modelling

BLOG Web Log

BSDMC_MM Bad Smell Detection and Metrics Calculation Metamodel

CBO Coupling Between Object Classes

CWM Common Warehouse Metamodel

CVS Concurrent Versions System

DSL Domain Specific Language

ECL Epsilon Comparison Language

EGL Epsilon Generation Language

EWL Epsilon Wizard Language

EMF Eclipse Modeling Framework

EML Epsilon Merging Language

EMOF Essential Meta Object Facility

EOL Epsilon Object Language

Epsilon Extensible Platform of Integrated Languages for mOdel maNagement

ETL Epsilon Transformation Language

EU European Union

EVL Epsilon Validation Language

FAQ Frequently Asked Questions

FiWC Field in Wrong Class

FLAME Formal Library for Aiding Metrics Extraction

FRiC Field Related in Class

Fujaba From UML to Java And Back Again

GMF Graphical Modeling Framework

GMT Generative Modeling Technologies

GUI Graphic User Interface

GReAT Graph Rewrite And Transformation

IDE Integrated Development Environment

ISO International Organization for Standardization

IT Information Technology

JMI Java Metadata Interface

JVM Java Virtual Machine

KM3 Kernel Meta Meta Model

LHS Left Hand Side

LLOC Logical Lines of Code

M2M Model to Model

M2T Model to Text

MDA Model Driven Architecture

MDD Model Driven Development

MDE Model Driven Engineering

MDR Meta Data Repository

MDSD Model Driven-Software Development

MWE Modeling Workflow Engine

MOF Meta Object Facility

MOLA MOdel transformation LAnguage

MOOD Metrics for Object-Oriented Design

MOOSE Metrics for Object-Oriented Software Engineering

MTF Model Transformation Framework

NOC Number of Children

oAW openArchitectureWare

OCL Object Constraint Language

OMG Object Management Group

OML Operational Mapping Language

OMT Object Modeling Technique

OO Object Oriented

PDM Platform Description Model

PIM Platform Independent Model

PSM Platform Specific Model

QMOOD Quality Model for Object-Oriented Design

QVT Query/View/Transformation

RFP Request For Proposal

RHS Right Hand Side

SMM Software Metrics Meta-Model

TDM Transformation Description Model

UML Unified Modeling Language

UML2 Unified Modeling Language version 2

VIATRA2 VIsual Automated model TRAnsformations

VM Virtual Machine

XMI XML Metadata Interchange

XML Extensible Markup Language

XSD XML Schema

XSLT Stylesheet Language Transformation

YATL Yet Another Transformation Language

Bibliography

[1] Airbus - Welcome to Airbus.com! Available online at http://www.airbus.com/en/. Last checked : 31.01.2010.

[2] alphaWorks : Model Transformation Framework. Available online at http://www.alphaworks.ibm.com/tech/mtf. Last checked: 06.01.2010.

[3] Apache Ant - Welcome. Available online at http://ant.apache.org/. Last checked : 13.02.2010.

[4] ATL 3.0.0 New and Noteworthy - Eclipsepedia. Available online at http://wiki.eclipse.org/ATL_3.0.0_New_and_Noteworthy. Last checked : 07.01.2010.

[5] ATL Basic Examples and Patterns. Available online at http://www.eclipse.org/m2m/atl/basicExamples_Patterns/. Last checked: 06.01.2010.

[6] ATL Language Troubleshooter. Available online at http://wiki.eclipse.org/ATL_Language_Troubleshooter. Last checked: 06.01.2010.

[7] ATL Project. Available online at http://www.eclipse.org/m2m/atl/. Last checked: 28.02.2010.

[8] ATL Standard Library - Eclipsepedia. Available online at http://wiki.eclipse.org/ATL_Standard_Library. Last checked: 06.01.2010.

[9] ATL Transformations. Available online at http://www.eclipse.org/m2m/atl/atlTransformations/#UML22Measure. Last checked: 06.01.2010.

[10] AtlanMod Partners. Available online at http://www.emn.fr/z-info/atlanmod/index.php/Partners. Last checked: 06.01.2010.

[11] Atlas Model Weaver. Available online at http://www.eclipse.org/gmt/amw/. Last checked: 06.01.2010.

[12] ATL/Developer Guide - eclipsepedia. Available online at http://wiki.eclipse.org/ATL/Developer_Guide. Last checked: 06.01.2010.

[13] atl_discution By Thread. Available online at http://atlanmod.emn.fr/www/atl_discussion_archive/. Last checked: 06.01.2010.

[14] ATL/Howtos. Available online at http://wiki.eclipse.org/index.php/ATL_Howtos. Last checked: 06.01.2010.

[15] AToM3 a tool for multi-formalism meta-modelling. Available online at http://atom3.cs.mcgill.ca/index_html. Last checked: 06.01.2010.

[16] BAE Systems. Available online at http://www.baesystems.com/. Last checked: 31.01.2010.

[17] CARROLL. Available online at http://www.carroll-research.org/uk/index.htm. Last checked: 06.01.2010.

[18] Compuware Corporation. Available online at http://www.compuware.com/. Last checked: 14.02.2010.

[19] Context & Objectives - INESS Project. Available online at http://www.iness.eu/spip.php?article2. Last checked: 31.01.2010.

[20] CZT - Welcome to CZT. Available online at http://czt.sourceforge.net/. Last checked: 06.01.2010.

[21] Delphi from Embarcadero | RAD Application Development Software. Available online at http://www.embarcadero.com/products/delphi. Last checked : 07.01.2010.

[22] Developer Resources for Java Technology. Available online at http://java.sun.com/. Last checked : 13.02.2010.

[23] Eclipse Community Forums: Epsilon. Available online at http://www.eclipse.org/forums/index.php?t=thread&frm_id=22. Last checked: 06.01.2010.

[24] Eclipse Community Forums: GMT (Generative Modeling Technologies). Available online at http://www.eclipse.org/forums/index.php?t=thread&frm_id=109. Last checked: 06.01.2010.

[25] Eclipse Community Forums: M2M(model-to-model transformation). Available online at http://www.eclipse.org/forums/index.php?t=thread&frm_id=23. Last checked: 06.01.2010.

[26] Eclipse Community Forums: M2T (model-to-text transformation). Available online at http://www.eclipse.org/forums/index.php?t=thread&frm_id=24&. Last checked: 06.01.2010.

[27] Eclipse M2M Proposal. Available online at http://www.eclipse.org/proposals/m2m/. Last checked : 09.01.2010.

[28] Eclipse Modeling - M2M - Operational QVT - downloads. Available online at http://www.eclipse.org/modeling/m2m/downloads/index.php?project=qvtoml. Last checked : 11.01.2010.

[29] Eclipse.org home. Available online at http://www.eclipse.org/. Last checked: 26.02.2010.

[30] EMF model query developer guide. Available online at http://help.eclipse.org/galileo/index.jsp?nav=/9. Last checked: 06.01.2010.

[31] Epsilon. Available online at http://www.eclipse.org/gmt/epsilon/. Last checked: 06.01.2010.

[32] Epsilon Model Connectivity. Available online at http://www.eclipse.org/gmt/epsilon/doc/emc/. Last checked: 06.01.2010.

[33] Fujaba Homepage. Available online at http://www.fujaba.de/. Last checked: 06.01.2010.

[34] Home, WesternGeco. Available online at http://www.westerngeco.com/. Last checked: 31.01.2010.

[35] HP United States - Computers, Laptops, Servers, Printers and more. Available online at http://www.hp.com/#Product. Last checked: 31.01.2010.

[36] http://www.ample-project.net/. Available online at http://www.ample-project.net/. Last checked: 31.01.2010.

[37] IBM Research - Zurich. Available online at http://www.zurich.ibm.com/. Last checked: 31.01.2010.

[38] ISIS repository - GReAT. Available online at http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT. Last checked : 13.01.2010.

[39] itemis AG - Model Based Software Development. Available online at http://www.itemis.com/. Last checked: 31.01.2010.

[40] Kermeta - Breathe life into your metamodels - Kermeta. Available online at http://www.kermeta.org/. Last checked: 06.01.2010.

[41] King's College London :Home :King's College London. Available online at http://www.kcl.ac.uk/. Last checked: 31.01.2010.

[42] Live. Available online at http://www.eclipse.org/gmt/epsilon/live/. Last checked: 06.01.2010.

[43] Members - AtlanMod. Available online at http://www.emn.fr/z-info/atlanmod/index.php/Members. Last checked: 06.01.2010.

[44] ModelMorf Homepage. Available online at http://121.241.184.234:8000/ModelMorf/ModelMorf.htm. Last checked: 07.01.2010.

[45] Modelplex website - Homepage. Available online at http://www.modelplex-ist.org/. Last checked: 06.01.2010.

[46] openArchitectureware tutorial. Available online at http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/emf_tutorial.html. Last checked: 06.01.2010.

[47] openarchitectureware.org - Official openArchitectureware Homepage. Available online at http://www.openArchitectureWare.org/index.php. Last checked: 06.01.2010.

[48] openarchitectureware.org - Official openArchitectureware Homepage. Available online at http://www.openarchitectureware.org/forum/index.php?forum=2. Last checked : 07.01.2010.

[49] openarchitectureware.org - openArchitectureware Charter. Available online at http://www.openarchitectureware.org/article.php/oaw_charter?menu=About_Charter. Last checked: 06.01.2010.

[50] openarchitectureware.org - Professional Support & Consulting. Available online at http://www.openArchitectureWare.org/staticpages/index.php/support?menu=Professional%20Support. Last checked: 06.01.2010.

[51] openarchitectureware.org - Screencasts. Available online at http://www.openarchitectureware.org/staticpages/index.php/oaw_screencasts. Last checked: 06.01.2010.

[52] OpenEmbeDD - OpenEmbeDD. Available online at http://openembedd.org/home_html. Last checked: 06.01.2010.

[53] Oracle 11g, Siebel, PeopleSoft | Oracle, Software. Hardware. Complete. Available online at http://www.oracle.com/index.html. Last checked: 31.01.2010.

[54] Project-ATLAS:Programme blanc ANR FLFS (2006-2009). Available online at http://ralyx.inrialpes.fr/2007/Raweb/atlas/uid39.html. Last checked: 06.01.2010.

[55] Projects - AtlanMod. Available online at http://www.emn.fr/z-info/atlanmod/index.php/Projects. Last checked : 09.01.2010.

[56] Siemens AG - Global Web Site. Available online at http://w1.siemens.com/entry/cc/en/. Last checked: 31.01.2010.

[57] SmartQVT - A QVT implementation. Available online at http://smartqvt.elibel.tm.fr/index.html. Last checked: 06.01.2010.

[58] Software Architecture Design, Visual UML & Business Process Modeling - from Borland. Available online at http://www.borland.com/us/products/together/index.html. Last checked: 06.01.2010.

[59] SouceForge.net: epsilonlabs. Available online at http://sourceforge.net/apps/mediawiki/epsilonlabs/index.php?title=Main_Page. Last checked: 06.01.2010.

[60] Team-AtlanMod:IdM++. Available online at http://ralyx.inria.fr/2008/Raweb/atlanmod/uid35.html. Last checked: 06.01.2010.

[61] Tefkat : The EMF transformation Engine. Available online at http://tefkat.sourceforge.net/. Last checked: 06.01.2010.

[62] Telefónica. Available online at http://www.telefonica.com/en/home/jsp/home.jsp. Last checked: 31.01.2010.

[63] Thales Group Home. Available online at http://www.thalesgroup.com/. Last checked : 31.01.2010.

[64] The <AGG> Homepage. Available online at http://user.cs.tu-berlin.de/~gragra/agg/. Last checked : 14.01.2010.

[65] The University of York. Available online at http://www.york.ac.uk/. Last checked: 29.01.2010.

[66] Topcased - Home. Available online at http://www.topcased.org/. Last checked: 06.01.2010.

[67] Université Pierre et Marie CURIE - Sciences et Médecine - UMPC - Paris. Available online at http://www.upmc.fr/. Last checked: 31.01.2010.

[68] Usine Logicielle - Detailed introduction. Available online at http://www.usine-logicielle.org/index.php?option=com_content&task=view&id=17&Itemid=30&limit=1&limitstart=1. Last checked: 06.01.2010.

[69] VIATRA2 - Eclipsepedia. Available online at http://wiki.eclipse.org/VIATRA2. Last checked : 13.01.2010.

[70] Welcome to NetBeans. Available online at http://netbeans.org/. Last checked: 06.01.2010.

[71] Xanium On-Demand Solutions. Available online at http://www.xactium.com/. Last checked: 22.01.2010.

[72] XSL Transformations (XSLT). Available online at http://www.w3.org/TR/xslt. Last checked: 06.01.2010.

[73] *openArchitectureware User Guide: Version 4.3.1*, December 2008. Available online at http://www.openArchitectureWare.org/pub/documentation/4.3.1/openArchitectureWare-4.3.1-Reference.pdf. Last checked : 20.01.2010.

[74] [galileo] Convert Xpand/Xtend projects from oaw4, July 2009. Available online at http://ekkescorner.wordpress.com/2009/07/23/galileo-convert-xpandxtend-projects-from-oaw4/. Last checked: 29.01.2010.

[75] QVTOML/Examples/InvokeInJava - Eclipsepedia. Available online at http://wiki.eclipse.org/QVTOML/Examples/InvokeInJava. Last checked: 06.01.2010, Last checked: 06.01.2010.

[76] F. Allilaire, J. Bézivin, F. Jouault, and I. Kurtev. ATL: Eclipse support for model transformation. 2006.

[77] M. Andersson and P. Vestergren. Object-Oriented Design Quality Metrics. Master's thesis, Uppsala University, Sweden, June 2004.

[78] ATLAS group LINA & INRIA, Nantes - France. *ATL: Atlas Transformation Language*, 2005. Available online at http://www.eclipse.org/m2m/atl/doc/ATL_VMSpecification[v00.01].pdf. Last checked: 20.01.2010.

[79] ATLAS group LINA & INRIA, Nantes - France. *KM3: Kernel MetaMetaModel*, August 2005. Available online at http://www.eclipse.org/gmt/am3/km3/doc/KernelMetaMetaModel%5Bv00.06%5D.pdf. Last checked: 20.01.2010.

[80] J. Bansiya and C. G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[81] A. L. Baroni. Formal definition of object-oriented design metrics. Master's thesis, Vrije Universiteit Brussel, 2002.

[82] A. L. Baroni and F. B. e Abreu. A Formal Library for Aiding Metrics Extraction. 2003.

[83] W. Brown, R. Malveau, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[84] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11):197 – 211, 1991.

[85] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[86] V. Cortellessa, S. di Gregorio, and A. di Marco. Using ATL for transformations in software performance engineering: a step ahead of java-based transformations? In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 127–132, New York, NY, USA, 2008. ACM.

[87] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE 2005 - Generative Programming and Component Enginering. 4th International Conference*, pages 422–437. Springer, 2005.

[88] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[89] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[90] S. Davy, B. Jennings, and J. Strassner. Application Domain Independent Policy Conflict Analysis Using Information Models. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 17–24, April 2008.

[91] Ł. Dobrzański. UML Model Refactoring: Support for Maintenance of Executable UML Models. Master's thesis, Blekinge Institut of Technologie, Sweden, July 2005.

[92] G. Doux, F. Jouault, and J. Bézivin. Transforming BPMN process models to BPEL process definitions with ATL. In *GraBaTs 2009 : 5th International Workshop on Graph-Based Tools*, 2009.

[93] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes. Engineering a DSL for Software Traceability. In *Software Language Engineering*, volume 5452/2009, pages 151–167, 2009.

[94] R. Dvorak. Model transformation with operation QVT. Borland Software Corporation, 2008.

[95] F. B. e Abreu. Design Metrics for Object-Oriented Software Systems. *European Conference on Object-Orietend Programming*, August 1995.

[96] T. Van Enckevort. Refactoring UML models: using openarchitectureware to measure UML model quality and perform pattern matching on UML models with OCL queries. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 635–646. ACM, 2009.

[97] F. Fondement and R. Silaghi. Defining Model Driven Engineering Processes. In *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML)*, 2004.

[98] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley Professional, third edition, September 2003.

[99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley Professional, 1999.

[100] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[101] D. Gasevic, D. Djuric, and V. Devedzic. *Model Driven Engineering and Ontology Development*. Springer-Verlag, 2009.

[102] I. Groher and M. Voelter. XWeave: Models and Aspects in Concert. 2007. Available online at `http://www.voelter.de/data/workshops/AOM2007.pdf`. Last checked : 27.01.2010.

[103] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse series. Addison-Wesley Professional, first edition, March 2009.

[104] V. Gruhn, D. Pieper, and C. Roettgers. *MDA: Effektives Software-Engineering mit UML2 und Eclipse*. Xpert.press. Springer, 2006.

[105] P. Huber. The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Master's thesis, TU Wien, May 2008.

[106] International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC). *ISO/IEC Standard No. 9126-1:2001(E)*, June 2001.

[107] A. A. Jalbani, J. Grabowski, H. Neukirchen, and B. Zeiss. Towards an Integrated Quality Assessment and Improvement Approach for UML Models. In *SDL 2009: Design for Motes and Mobiles*, volume 5719/2009 of *Lecture Notes in Computer Science*, pages 63–81. Springer Berlin / Heildelberg, 2009.

[108] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like Transformation Language. October 2006.

[109] F. Jouault and I. Kurtev. Transforming models with ATL. 2005.

[110] F. Jouault and I. Kurtev. On the interoperability of model-to-model transformation languages. In *Science of Computer Programming*, volume 68, pages 114–137. Elsevier North-Holland, Inc., 2007.

[111] Frédéric Jouault, editor. *Model Transformation with ATL*. 1st International Workshop, MtATL 2009 - Preliminary Proceedings, July 2009.

[112] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language MOLA. In *Model-Driven Architecture: Foundations and Applications*, pages 14–28, 2004.

[113] N. Klein and R. Nikonowicz. Wiederverwendbare Generatorkomponenten in heterogenen projektumfeldern. In *Software Engineering 2009*, pages 113–123, Kaiserslautern, March 2009.

[114] D. S. Kolovos. Dimitris Kolovos: Home Page. Available online at `http://www-users.cs.york.ac.uk/~dkolovos/`. Last checked: 06.01.2010.

[115] D. S. Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, 2008.

[116] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. In *Workshop on Model Driven Tools and Process Integration (MDTPI) EC-MDA '08*, 2008.

[117] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon Trasnformation Language. In *Theory and Practice of Model Transformations*, volume 5063/2008 of *Lecture Notes in Computer Science*, pages 36–60. Springer Berlin / Heildelberg, 2008.

[118] D. S. Kolovos, L. Rose, R. F. Paige, and F. A. C. Polack. *The Epsilon Book*. September 2009.

[119] B. P. Lamancha, P. R. Mateo, I. R. de Guzmán, M. P. Usaola, and M. P. Velthius. Automated model-based testing using the UML testing profile and QVT. In *MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pages 1–10. ACM, 2009.

[120] M. López-Sanz, J. M. Vara, E. Marcos, and C. E. Cuesta. A model-driven approach to weave architectural styles into service-oriented architectures. In *MoSE+DQS '09: Proceeding of the first international workshop on Model driven service engineering and data quality and security*, pages 53–60, New York, NY, USA, 2009. ACM.

[121] S. Marković and T. Baar. Semantics of OCL specified with QVT. In *Software Systems Modeling*, volume 7, pages 399–422. Springer Berlin / Heildelberg, October 2008.

[122] T. Mens and T. Tourwé. A Survay of Software Refactoring. In *IEEE Transactions on Software Engineering*, volume 30, pages 126–139. IEEE Press, 2004.

[123] S. Nolte. *QVT - Relations Language : Modellierung mit der Query Views Transformation*. Xpert.press. Springer-Verlag Berlin Heidelberg, first edition, 2009.

[124] S. Nolte. *QVT - Operational Mappings : Modellierung mit der Query Views Transformation*. Xpert.press. Springer-Verlag Berlin Heidelberg, first edition, 2010.

[125] Object Management Group (OMG). *MDA Guide Version 1.0.1*, June 2003. Available online at `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`. Last checked : 20.01.2010.

[126] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification, Version 2.0*, January 2006. Available online at `http://www.omg.org/spec/MOF/2.0/PDF/`. Last checked : 20.01.2010.

[127] Object Management Group (OMG). *MOF 2.0/XMI Mapping, Version 2.1.1*, December 2007. Available online at `http://www.omg.org/spec/XMI/2.1.1/PDF/`. Last checked : 20.01.2010.

[128] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification*, April 2008. Available online at `http://www.omg.org/spec/QVT/1.0/PDF/`. Last checked : 20.01.2010.

[129] Object Management Group (OMG). *Architecture-Driven Modernization (ADM): Software Metrics Meta-Model FTF - Beta 1*, March 2009. Available online at `http://www.omg.org/spec/SMM/1.0/Beta1/PDF`. Last checked : 20.01.2010.

[130] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML), Infrastructure. Version 2.2*, February 2009. Available online at `http://www.omg.org/cgi-bin/doc?formal/09-02-04.pdf`. Last checked : 20.01.2010.

[131] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML) Superstructure Version 2.2*, February 2009. Available online at `http://www.omg.org/cgi-bin/doc?formal/09-02-03.pdf`. Last checked : 20.01.2010.

[132] Object Management Group (OMG). *Object Constraint Language, Version 2.2*, February 2010. Available online at `http://www.omg.org/spec/OCL/2.2/`. Last checked : 14.02.2010.

[133] O. Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.

[134] R. Petrasch and O. Meimberg. *Model Driven Architecture: Eine praxisorientierte Einführung in die MDA*. dpunkt.verlag GmbH, first edition, 2006.

[135] G. Pietrek and J. Trompeter (Hrsg.). *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis*. Entwickler.press, first edition, 2007.

[136] E. Riccobene and P. Scandurra. Weaving executability into uml class models at pim level. In *BM-MDA '09: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 1–9, New York, NY, USA, 2009. ACM.

[137] L. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack. Constructing Models with the Human-Usable Textual Notation. In *Model Driven Engineering Languages and Systems*, volume 5301/2009 of *Lecture Notes in Computer Science*, pages 249–263, 2009.

[138] L. H. Rosenberg. Applying and interpreting object oriented metrics. Available online at `http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html`. Last checked: 06.01.2010.

[139] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley Professional, second edition, 2004.

[140] M. Sarker. An overview of Object Oriented Design Metrics. Master's thesis, Umeå University, Sweden, June 2005.

[141] J. M. Spivey. *The Z Notation: A Reference Manual, second edition*. Second edition, 1998. Available online at `http://spivey.oriel.ox.ac.uk/mike/zrm/zrm.pdf`. Last checked : 20.01.2010.

[142] T. Stahl and M. Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, first edition, June 2006.

[143] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag, 2007.

[144] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. The Eclipse series. Addison-Wesley, second edition, January 2009.

[145] G. Sunyé, D. Pollet, Y. L. Traon, and J. Jézéquel. Refactoring UML Models. In *UML 2001 - The Unified Modeling Language - Modeling Languages, Concepts, and Tools*, volume 2185-2001 of *Lecture Notes in Computer Science*, pages 134–148. Springer Berlin / Heildelberg, 2001.

[146] N. Tsantalis and A. Chatzigeorgiou. Identification of Move Method Refactoring Opportunities. In *IEEE Transactions on Software Engineering*, volume 35, pages 347–367, May-June 2009.

[147] B. Veliev. Model-to-Model transformationen in openArchitectureWare. In *Transformationen in der modellgetriebenen Software-Entwicklung*, pages 82–100. Universität Karlsruhe – Fakultät für Informatik, 2007.

[148] C. Werner. *UML Profile for Communication Systems : A New UML Profile for the Specification and Description of Internet Communication and Signaling Protocols*. PhD thesis, Georg-August-Universität zu Göttingen, 2006.