A Performance Analysis of Model Transformations and Tools

Steven Bosems

A Performance Analysis of Model Transformations and Tools

THESIS

to obtain the degree of Master of Science
on Thursday March 24, 2011
Department of Electrical Engineering Mathematics and Computer Science
University of Twente

by

Steven Bosems
born on December 3, 1985
in Apeldoorn, The Netherlands

This thesis has been approved by:
Dr. I. Kurtev
Dr. L. Ferreira Pires

Acknowledgments

Over the last few months, I have received a lot of help and support by many people whom I would like to thank here.

I would like to start by thanking my supervisors Ivan and Luís. They have been a source of inspiration for me throughout the process of the research and writing. Their feedback and insights were always valuable, and never went unused. Writing a paper with them was a new experience, but they were patient with me.

Related to this, I would like to thank Marcel van Amstel. Without his knowledge on metrics, my performance measures would have been just numbers.

Special thanks go to Klaas van den Berg. Although not directly involved in this graduation project, he did supervise my research that was the basis of it.

My roommates at the fifth floor at Zilverling provided a great working environment. I thank them for the laughs and talks we had.

Not all was work during the past few months. Therefore, I would like to thank my friends at ZPV-Piranha for the diving trips we made together, both in the murky waters in The Netherlands and in the deep blue seas around Curaçao.

Of course, this acknowledgment would complete without thanking my parents. Having supported me throughout my university, I cannot express my gratitude enough.

My girlfriend Renske more than deserves to be named here too. Throughout the process of my graduation research, she has been loving and supportive.

Steven Bosems
March 15, 2011

Abstract

Model-Driven Engineering is a software development process that has gained popularity in the recent years. Unlike traditional software engineering processes, MDE is centered around models, instead of code. By using model transformations, models can be translated from one language to another, resulting in a separation of program architecture and execution platform. However, an increase in size of any of the elements required by the transformation process might lead to performance problems. Although these problems are common and well known in the field of software engineering, problems specific to MDE have not yet been investigated in sufficient depth.

In this research, we compare the performance of three model transformation engines. These tools allow the transformation of models to be specified in ATL, QVT Operational Mappings and QVT Relations. Furthermore, different implementation strategies are evaluated to determine how language constructs affect the performance of the model transformation process.

The implementation of model transformation engines determines the performance of the language. Increases of model size and complexity cause transformations to run slower, yet some transformation engines are affected more than others. ATL is the fastest performing language, followed by QVTo and QVTr in this order.

Language constructs often allow developers to define the same model transformation in multiple ways. High metric values for the number of attribute helpers, and low values for the number of calls to `allInstances()` indicate better performance in ATL transformations. High values for the number of called rules metric suggests an imperative specification style, resulting in a negative impact on performance.

The results from this research allow transformation designers to estimate the performance of their transformation definitions. Developers of model transformation tools can use our results to improve the current version of their tools.

Contents

Acknowledgments **i**

Abstract **iii**

Glossary **xi**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem statement 2
1.3 Objectives and approach 2
1.4 Contributions 3
1.5 Outline 3

2 Model-Driven Engineering **5**
2.1 History and definition 5
2.2 MDE elements 6
2.3 Conclusion 8

3 Transformation languages and tools **11**
3.1 ATLAS Transformation Language 11
3.2 Query/View/Transformations Relations language 14
3.3 Query/View/Transformations Operational Mappings language 15
3.4 Comparison 16
3.5 Conclusion 16

4 Metrics in MDE **19**
4.1 Size 19
4.2 Complexity 23
4.3 Problems 27
4.4 Conclusion 29

5 Transformation experiments **31**
5.1 Model transformations 31
5.2 Experiments 32
5.3 Execution 34
5.4 Transformation explanations 34
5.5 Explanation on metrics 35
5.6 SimpleClass2SimpleRDBMS 36
5.7 RSS2ATOM 39
5.8 Analysis 40
5.9 Discussion 41
5.10 Conclusion 42

6 Conclusion **43**
6.1 Model-Driven Engineering 43
6.2 Transformation languages and tools 44
6.3 Metrics in MDE 44
6.4 Transformation experiment 45
6.5 Evaluation 45
6.6 Future work 45

References **47**

A Transformation results **51**
A.1 Metrics 51
A.2 SimpleClass2SimpleRDBMS 54
A.3 RSS2ATOM 57

B Model generation code **59**
B.1 SimpleClass model generation 59
B.2 RSS model generation 62

List of Figures

2.1 The metamodeling stack 7
2.2 Metamodeling stack with transformations 9

5.1 SimpleClass2SimpleRDBMS, varying classes 37
5.2 SimpleClass2SimpleRDBMS, varying attributes 38
5.3 RSS2ATOM 40

List of Tables

3.1 Comparison of transformation languages 17

A.1 SimpleClass2SimpleRDBMS metamodel metrics 52
A.2 RSS2ATOM metamodel metrics 52
A.3 Metrics for the ATL transformations 53
A.4 Metrics for the QVTo transformations 53
A.5 Initial ATL transformation (general case) 54
A.6 Initial ATL transformation (worst case) 54
A.7 ATL_a transformation (general case) 54
A.8 ATL_a transformation (worst case) 55
A.9 ATL_b transformation - Language (general case) 55
A.10 ATL_b transformation - Language (worst case) 55
A.11 QVT Operational Mappings (QVTo) (general case) 55
A.12 QVTo (worst case) 56
A.13 QVT Relations (QVTr) (general case) 56
A.14 QVTr (worst case) 56
A.15 RSS2ATOM transformation 57

Glossary

ATL The ATLAS Transformation Language, which is generally used to "express MDA-style model transformations, based on explicit metamodel specifications" [5].

GQM The Goal/Question/Metrics approach can be used to find metrics for given question, which is derived from an overall goal. The GQM method was proposed by Solingen and Berghout [33].

instanceOf relation "Strictly speaking instanceOf relation denotes a membership of an entity to a class where the class is a set." [22]

Language "Let Σ be an alphabet [and λ the empty string]. Σ^*, the set of strings over Σ, is defined as follows:

1. Basis: $\lambda \in \Sigma^*$.
2. Recursive step: If $w \in \Sigma^*$ and $a \in \Sigma$, then $wa \in \Sigma^*$.
3. Closure: $w \in \Sigma^*$ only if it can be obtained from λ by a finite number of applications of the recursive step.

For any nonempty alphabet Σ, Σ^* contains infinitely many elements. [...]
A language over an alphabet Σ is a subset of Σ^*." [35]
In other words: a language is a set of words over a finite alphabet.

Metamodel "A metamodel is a model of a modeling language." [22]

Model "A model represents a part of the reality called the object system and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system." [22]

Model transformation "A model transformation is a process of automatic generation of a target model from a source model, according to a transformation definition, which is expressed in a model transformation language." [22]

Module A module is a set of rules, helpers and attributes that combined form a transformation definition. ATL uses the term "module", both QVT languages use "transformation".

Program A program is an instance of a language grammar.

QVT The Query/View/Transformation specification was proposed by the Object Management Group [28]. It consists of three languages: QVT Core, QVT Relations and QVT Operational Mappings.

Chapter 1

Introduction

This chapter gives an introduction to this thesis. It discusses the objectives and approach of this research, lists the contributions made to the field of Model-Driven Engineering and gives an outline of this thesis.

1.1 Motivation

Since their first introduction, computers have become powerful machines, which are able to run millions of calculations per second. Application programmers and users alike have taken full advantage of this, requesting and adding more features to their programs than ever before. As a result, computer programs have grown, making maintainability increasingly troublesome. In order to circumvent these problems, new techniques and methods are introduced continuously, aiming to relieve developers from these difficulties.

A trend that started with the introduction of FORTRAN is that development techniques have raised the level of abstraction from code, moving from code that could be understood best by machines, to code that was understandable to humans. In this way, programmers no longer needed to think in terms of ones and zeros running on the computer hardware, rather, they could focus on creating solutions for the problems at hand.

Currently, software development techniques focus even less on the programming code. Instead, models have become increasingly important. Because models can be created in a platform-independent way, they can still be used even if the underlying

implementation techniques are changed. This software development approach is known as Model-Driven Engineering (MDE) [21]. MDE is defined as a framework that can be used to specify software development processes that see models as their primary artifact.

Model transformations play an important role in MDE. As such, our research primarily focuses on these MDE artifacts. We use the following definition for a model transformation:

> "A model transformation is a process of automatic generation of a target model from a source model, according to a transformation definition, which is expressed in a model transformation language."[22]

1.2 Problem statement

When performing a model transformation, run by a given *transformation engine*, we generate an *output model* from an *input model*, both of which are instances of *metamodels*.

However, a situation faced by software developers as artifacts become larger, is that this generation may either take a long time, or can cause the system as a whole to stop responding, much like an ordinary computer program with large input. The exact cause for this is unknown, since metamodel-based transformation techniques are recent and more study is required to get a better understanding of their working. These problems may limit the use of MDE techniques in practice.

1.3 Objectives and approach

The objective of this research can be stated as follows:

> *"Investigate the performance of model transformations and transformation languages and evaluate the effects of alternative model transformation implementation strategies"*

To fulfill this objective, the following steps have been performed:

Define size and complexity in the domain of Model-Driven Engineering. In order to be able to analyze the performance of model transformations, we have defined metrics for MDE. These metrics have been used to quantify the size and complexity of these elements. They allow us to determine which properties of MDE elements potentially affect transformation performance.

Study and compare different transformation language implementations. We have studied and compared three popular transformation languages: the ATLAS Transformation Language, the Query/View/Transformations Operational Mappings language

and the Query/View/Transformations Relations language. Two transformations have been used for this purpose: the SimpleClass2SimpleRDBMS transformation was chosen for its graph-like structure, the RSS2ATOM transformation was selected because of its tree structure.

Study and compare the effect of alternative transformation definition implementations. The hybrid transformation language ATL allowed us to implement equivalent transformation definitions using different language constructs. We have studied the effects of these alternative implementations on the performance of the transformation execution.

1.4 Contributions

This thesis makes the following contributions to the field of Model-Driven Engineering:

An overview of size and complexity metrics in the field of Model-Driven Engineering. Size and complexity metrics for MDE artifacts are explored and discussed. These can be used to predict the quality and performance of elements in an MDE process.

A performance analysis of model transformation engines. The performance of three model transformation engines is measured. This will provide insight in the differences among model transformation tools.

A performance comparison of model transformation implementations. One ATL model transformation is implemented using three implementation strategies. Using metrics, this will allow developers to estimate the performance of their model transformations.

1.5 Outline

This thesis has the following structure:

Chapter 2 - Model-Driven Engineering introduces the concept of Model-Driven Engineering in order to provide the background required for the rest of this thesis. The different elements of MDE are discussed and related to each other.

Chapter 3 - Transformation languages and tools gives an overview of three popular model transformation languages and tools: ATL, QVTr and QVTo. The features of each language are evaluated, and the tools are compared based on their implementation. These tools are used to perform our model transformation experiment.

Chapter 4 - Metrics in MDE explores existing metrics that can be incorporated to measure MDE artifacts. These measurements are used to analyze the results obtained from our experiments.

Chapter 5 - Transformation experiments presents the setup and results of our experiments. The transformations performed are explained and we discuss why these transformations were chosen. An analysis of the results is reported and discussed.

Chapter 6 - Conclusion summarizes our findings and suggests topics for future research.

Previous publications

Parts of this work have previously been published in [43].

Chapter 2

Model-Driven Engineering

Models have become increasingly important to design and document software applications. MDE takes this view one step further: in MDE, every artifact is a model. In this chapter, we discuss the elements of MDE, exploring the metamodeling stack and an example of an MDE process. This knowledge is used in the rest of this research.

2.1 History and definition

In 2003, the Object Management Group (OMG) officially introduced the Model-Driven Architecture (MDA) [25]. The specification explains how different OMG standards could be used together. MDA focuses on the concepts of Platform Independent Models and Platform Specific Models, two viewpoints on software systems, and how mappings between these two can be made in order to streamline software development. Through this approach, the functional specification of the system and the implementation specification are separated, allowing for better reuse and portability. Although it is noted that this mapping can be automated, this is not discussed.

Kent [21] reviewed the MDA, and defined how the techniques described by the OMG can be used in order to obtain a complete process. This process relies on models as the primary artifact. Kent explains how tools play an important role in the process he refers to as Model-Driven Engineering (MDE). The author argues that these tools can help developers to maintain different levels of models. The resulting process revolves

around metamodeling and metamodel-based transformations. These transformations can either be model transformations or language transformations. The first deals with the mapping between different model levels or viewpoints of a system, while the second performs a translation between languages.

2.2 MDE elements

2.2.1 Metamodeling stack

A metamodel is defined by [22] as follows:

> “A metamodel is a model of a modeling language.”

A model, in turn, is defined by [22] as:

> “A model represents a part of the reality called the object system and is expressed in a modeling language. A model provides knowledge for a certain purpose that can be interpreted in terms of the object system.”

Within the MDA [25], the Meta Object Facility (MOF) [27] is recommended as the main technology for defining modeling languages. MOF specifies four modeling levels:

- M3 Metametamodel
- M2 Metamodels
- M1 Models
- M0 Model instance

In the UML 2.0 Infrastructure Specification [30], the authors give meaning to the different levels of this metamodeling stack. The stack is visualized in Figure 2.1, as adopted from [22]. The layers are structured in a hierarchy, each level being an instance of the immediate level above it.

Level *M3* is the highest level, describing the metametamodel. This layer defines the language that is used to specify metamodels. OMG presents the MOF for this purpose.

The second layer, denoted as *M2*, is responsible for defining a language for the models. Metamodels at this level are instances of the metametamodel. A metamodel may specify a general purpose modeling language, like the Unified Modeling Language, or can be used to define a domain-specific language that is tailored to a certain application domain.

The third layer, named *M1*, contains instances of metamodels. Models at this level are also called user models: these models are commonly created by the end-user.

The last layer of the metamodeling hierarchy is *M0*. As [22] describes, *M0* does not contain models, instead it encompasses runtime instances, which are situated in the real world. These instances of models at level *M0* should, as such, not be regarded as models, and thus fall outside the scope of this thesis.

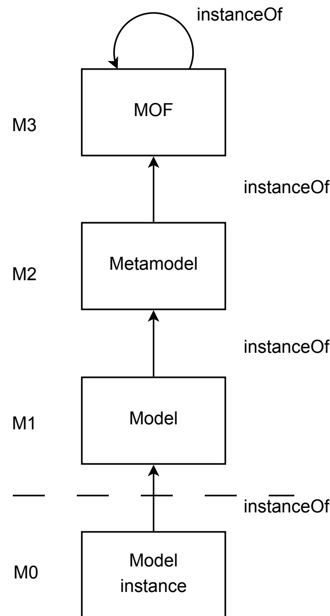


Figure 2.1: The metamodeling stack - This stack shows how the different MOF levels are related

2.2.2 Language

We define a language to be a set of strings over an alphabet [35]. Languages can be divided into two groups: natural and artificial languages. Natural languages include those languages that come to exist without being designed. In contrast, artificial languages are designed for a specific purpose. Computer languages, in which we are interested in this research, are created in order to allow programmers to solve problems using algorithms.

2.2.3 Model

The term model is broad, especially in the domain of MDE, where every artifact is a model [21]. Traditionally, a model is a simplified representation of a system [31]. In the domain of metamodel-based modeling, a model is an instance of a metamodel. However, a metamodel and a metametamodel are models too. A metametamodel is then an instance of itself. The model at this level is thus *self-reflective* [22].

2.2.4 Program

Since a program is an instance of a language grammar, and a language is a possible infinite amount of words, we can see a program as a collection of words that adheres to the restrictions as set by the language description.

When we look at the size of a computer program, we can identify two types of size and complexity: static and a dynamic size and complexity. The static properties of a program are those that are determined at design time, meaning that they exist in the realm of concrete files and folders on a hard drive. These files then contain lines of code. Dynamic properties of a program are time and space complexity of the running program, where resources include processor time and memory respectively. Since running programs are located in the real world, we are not interested in analyzing this type of size and complexity; rather, we focus on the code used to instantiate these programs.

2.2.5 Transformation

An important technique used in MDE is model transformation. Allowing programmers to specify mappings between two, or possibly more, metamodels, model transformations can increase productivity and quality, since a transformation has to be written only once, but may be performed an infinite amount of times. We observe that a transformation documents design decisions with relation to the mapping between metamodel elements.

2.2.6 Transformation process

We can now relate the different elements of MDA to each other. This is depicted in Figure 2.2.

This figure illustrates how the model transformation process is structured. The *user model* on the left, which is an instance of a *modeling language metamodel*, is used as an input for a *transformation*. This transformation is derived from a *transformation definition*, which in turn is an instance of a *transformation language*. The result of the process is a *software application*.

The language used to describe the transformation definition is specialized for this purpose. Examples of specialized languages are ATL, QVTr and QVTo. Unlike general purpose programs, in which the programming code is to be executed imperatively, model transformation definitions are models, defining mappings between the input and the output models. The process of transforming the input model through the use of this model transformation is performed by a model transformation engine.

The figure as a whole depicts the transformation from a model into another model, which may have a different metamodel. This is called a *model-to-model transformation*. Another type of transformation transforms a model into programming code. This type of transformation is referred to as a *model-to-text transformation*.

2.3 Conclusion

In this chapter, we have explored the basic concepts of Model-Driven Engineering. We have discussed the elements of the metamodeling stack, their relationships, and the role of model transformations in the MDE process. We have also discussed the four primary

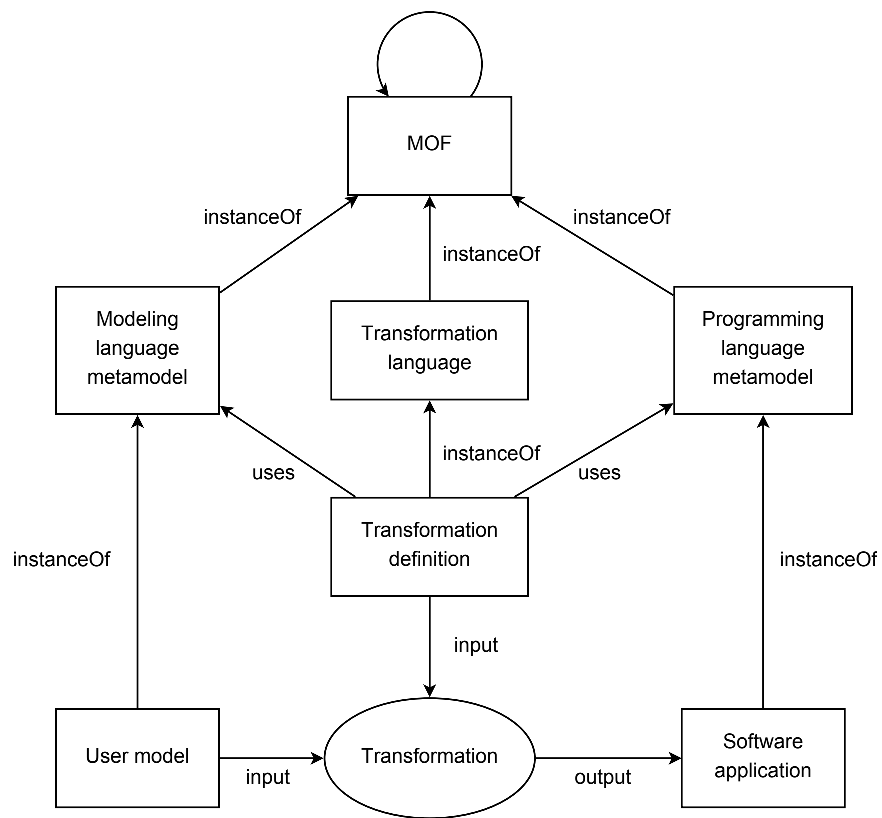


Figure 2.2: Metamodeling stack with transformations - The metamodeling stack, combined with model transformations

elements of the MDE process: languages, models, programs, and model transformations. This knowledge is used in the following chapters to obtain a better understanding of the performance of model transformations.

Chapter 3

Transformation languages and tools

Transformation languages are designed to generate an output model, given an input metamodel, an input model, and an output metamodel. Throughout the years, numerous projects have implemented their ideas on how a transformation language ought to look like. Three popular transformation languages are the ATLAS Transformation Language, the Query/View/Transformations Relations language and the Query/View/-Transformations Operational Mappings language. Created by INRIA and the OMG respectively, they incorporate many primary concepts used by transformation languages. In this chapter, we shall discuss these three languages, their commonalities and their differences in order to get a better understanding of their workings. These languages are used to perform our transformation experiments.

3.1 ATLAS Transformation Language

The ATLAS Transformation Language (ATL) allows users to define model-to-model transformations in both a declarative and an imperative way [37]. ATL was created by the Institut National de Recherche en Informatique et en Automatique (INRIA) as an answer to the Object Management Group's Query/View/Transformations language request for proposals [29]. OMG envisioned this language to perform the transformations in its Model-Driven Architecture (MDA) [28].

ATL is a hybrid language, allowing the developer to write transformations in both a declarative and an imperative style; the former is preferred, however, the latter can be used to express transformations that are otherwise hard to define.

In the style of MDE, ATL considers a *module* as a model of a model transformation. This module describes how to create the target models from the source models.

A transformation can be performed in *normal mode* or in *refining mode* [18, 37]. When in normal mode, the default mode of operation when executing a transformation, only those fields specified in the definition are mapped from the source to the target model. Source models are read-only, target models are write-only. This security feature prevents altering the source model, prohibiting the use of a model as both the source and the target of a transformation in an in-place update. In the refining mode, source and target models can be the same. Source elements that are not matched are copied automatically to the target model.

An ATL developer can specify his transformations using three types of *rules*: matched rules, lazy rules and called rules.

Matched rules	This is the primary type of rule to be used in declarative ATL transformations. It specifies which source element is to be matched, along with the target element that is to be generated.
Lazy rules	These rules are similar to matched rules, however, they are not executed when matched; rather, they rely on being called by another rule.
Called rules	Are rules that are used when describing transformations imperatively. A called rule, like a matched rule, can generate target model elements. However, unlike a matched rule, a called rule can have parameters that are passed as the rule is executed. Called rules should be called from imperative code.

In order to increase code reuse and facilitate navigation on models, ATL allows *helpers* to be written. Being described as equivalent to Java methods [37], helpers contain ATL expressions that may be called from an arbitrary place in the transformation. Helpers are side-effect free.

Attribute helpers are similar to helpers, however, they do not accept arguments. As a result, an attribute helper can be seen as a constant. The right hand side of an attribute definition is calculated once, after which the result is stored for future reference.

Like helpers, *queries* can be specified to aid the computation of the output model. A query is a transformation from a model element to a primitive type value, so that the result of a query is most often a string, numerical or boolean value.

ATL *libraries* are reusable units which can be called from different ATL modules, queries and other libraries. Libraries, unlike rules, helpers and queries cannot be executed independently.

3.1.1 Eclipse M2M ATL plug-in

The ATL architecture is explained in [36]. It consists of four parts: the Core, Parser, Compiler and the Virtual Machine. The ATL IDE [3] is implemented on the Eclipse platform.

Core ATL Core consists of several services and classes, that, when combined, drive the model transformation. Two services are provided: the CoreService and LauncherService. The CoreService is a utility that allows other classes to perform ATL related tasks within the Eclipse framework. The LauncherService allows the user to launch a transformation, given metamodels and model names.

The classes provided by Core are used to internally represent a model (IModel), assume the role of metamodels (IReferenceModel), allow creation of models and metamodels (ModelFactory), provide ways to load and save models created using the ModelFactory (IInjector and IExtractor) and supply ways to launch a transformation (ILauncher). All these classes are used by the LauncherService in order to perform the model transformation, given launch configurations and a set of ant tasks.

Parser The ATL Parser has an ATL transformation definition as an input, and outputs an ATL model that to the ATL metamodel. Using well-formedness rules, this model is then used as an input for the generation of a problem model, which can be used by the editor to insert markers in the code.

Compiler Since its 2006 version, the ATL compiler uses the ATL VM Code Generator (ACG). ACG is a domain-specific language (DSL), aimed at the compilation and generation of code. Its abstract syntax is defined in a KernelMetaMetaModel (KM3) [17], and the concrete syntax is specified in the Textual Concrete Syntax (TCS) language [19].

Since the compiler is a DSL, it does not work on an imperative language basis. Instead, it accepts a model of the content of the compiled file. Before execution, the compiler is bootstrapped by an `ACG.acg` file, containing the description of the ACG compiler. After this, an ASM file is generated from the input by traversing it using a visitor pattern. The ASM file is used as input for the Virtual Machine.

Virtual Machine As in most Virtual Machines (VMs), the ATL Virtual Machine is a byte code interpreter [4]. The input for the VM is an ASM file, together with the input model. The input model is read by a Model Handler, after which it is handed to the execution environment. This environment executes all operations defined by the input sequentially based on the type of operation. During execution, the output model is generated and serialized.

3.2 Query/View/Transformations Relations language

The Query/View/Transformations (QVT) Relations language was proposed by the Object Management Group (OMG) in 2005 [28]. In 2008 the specification was finalized and released [29] as an adopted version.

QVTr was created as a user-friendly language that would allow developers to specify their model transformations in a declarative way, defining relationships between MOF models [29].

Transformations in QVTr are defined as a set of relations between the source and the target model, which must conform to their respective metamodels. Once all relations hold, the transformation is successful. The transformation fails when not all relations can be satisfied. Relations may be enforced by modifying the target such that it holds. The target model may also be created when it is empty. In contrast, a relation may also be used to check if the relationship is satisfied without resulting in any alteration of the target model.

By using *when* and *where* clauses, a relation may further be restricted. The transformation will succeed only if the other relations succeed too in case of the *when* clause. In case of the *where* clause, the other relations need to succeed as a requirement for the success of this relation.

A transformation as a whole can only succeed if all its *top-level relations* succeed. These relations are always executed, which is not the case with non-top-level relations, which are only executed when called from a *where* clause.

3.2.1 medini QVT

Since QVT Relation is provided as a standard, vendors have to implement this standard in order to create programs. One of these vendors is [14], which has developed medini QVT. This tool is written as a plug-in for the Eclipse platform.

Unlike the M2M ATL implementation, medini QVT does not compile the QVT source to byte-code, rather, it acts as an interpreter. The transformation engine consists of a Parser, an Analyzer, an Evaluator, and a Visitor.

Parser The QVT Parser reads a given QVTr transformation and parses its contents. The result is a Concrete Syntax Tree (CST) of the given transformation. This CST is used as the input for the next phase.

Analyzer By using the CST generated by the Parser, the QVT Analyzer builds an Abstract Syntax Tree (AST) for the transformation. The Analyzer verifies that the transformation is syntactically correct, which is not done by the Parser.

Evaluator The Evaluator takes the AST generated in the previous phase and generates a data map for it. The data map contains the traces of the transformation, which can be used by the next phase to execute the transformation.

Visitor In the final phase, the Visitor takes the data map and starts executing the relations defined in the original transformation. In this phase, the actual mappings are performed, along with the generation of the target models and the trace file. After this, the output model and trace file are serialized.

3.3 Query/View/Transformations Operational Mappings language

QVTr facilitates the developer with a declarative language to perform model transformations, while the QVT Operational Mappings (QVTo) language allows the user to implement them in an imperative way [29]. OMG provides this language to allow transformations to be created, even when a pure declarative specification is too difficult to write.

Unlike QVTr transformations, which, once specified, can be executed in both directions, the QVTo transformations are specified unidirectionally. The transformation is an instantiable entity with properties and operations.

QVTo, unlike QVTr, allows the use of *modeltypes*. A modeltype may either be strict or effective conformant to the metamodel specified. Strict conformance entails that the model is an instance of a metamodel, as is always the case with a QVTr transformation. However, an effective conformant model does not need to be an instance of the metamodel. Instead, a model is also valid when its structure resembles that of the metamodel. As a result, a model may still be used, even though the version of the metamodel has changed.

Like general-purpose languages, QVTo supports the use of *libraries*. Libraries may either be accessed or extended, resulting in an import or a combination of an import and inheritance respectively.

Where mappings are performed implicitly when regarding QVTr, QVTo requires them to be called explicitly. Initialization and closing code have to be written manually by the developer. However, similar to QVTr, in QVTo mappings, like relations, may contain *when* and *where* clauses.

Like ATL, QVTo allows *helpers* to be defined. They are structured like helpers in ATL, consisting of expressions that are sequentially executed [29]. Helpers can have side-effects on their parameters, altering them such that this alteration is visible after the execution of the helper body. *Queries* are helpers that do not have side-effects, much like ATL's attributes.

Finally, QVTo allows the user to call external libraries by using Black-box operations. These can be written in a different language, for example Java. This makes integration with existing libraries possible, as well as providing an interface to other languages.

3.3.1 Eclipse M2M QVTo plug-in

Originally created by Borland, the QVTo transformation engine is now part of the Eclipse Model-to-Model project. This project also hosts the ATL engine discussed in section 3.1.1. Like the QVTr transformation engine, the QVTo engine is an interpreter, consisting of the same components with their respective functions.

3.4 Comparison

Based on the overview of the three different transformation languages and their respective tool implementations, these languages and tools can be compared. This has already partially been done [16]. Our comparison can be found in Table 3.1.

One thing all implementations have in common, is that all projects are implemented in the Eclipse environment. As a result, they share a common library that is used to load and save metamodels, models and traces. However, these are processed quite differently by the tools: ATL is a hybrid transformation language, QVTo only allows imperative constructs, and QVTr can only cope with declarative transformations. This is, however, not a drawback, since it allows the QVTr language to perform its transformations bidirectionally: once relations have been specified, the transformation can be performed both in the direction of the initial target and the other way around, allowing the user to keep the input and output models in sync. ATL and QVTo do not allow for this behavior, requiring the user to explicitly define mappings for both directions.

All three languages have a facility to perform side-effect free operations, however, not all allow the user to specify operations that have side-effects. QVTr is guaranteed to be side-effect free, ATL and QVTo do allow side-effects to occur.

When considering trace generation, we notice that all languages support this feature. The same holds for the cardinality of the transformations, since all languages allow M-to-N transformations to be performed.

The way the transformation engines work is quite different among the three projects. The ATL engine first compiles the transformation to virtual machine byte code, which is executed by the Java Virtual Machine at runtime. medini QVT and the QVTo implementation of the M2M project do not perform this compilation step, rather they interpret the transformation on-the-fly.

3.5 Conclusion

ATL, QVTr, and QVTo are all model transformation languages. As such, their basic feature sets are quite similar, but they differ on key points. ATL is a hybrid language, which compiles its transformations to byte code, QVTr is a declarative language that allows bidirectional transformations to be created, and QVTo is an interpreted imperative language.

	ATL	QVTr	QVTo Mappings
Implementation	Eclipse Plug-in	Eclipse Plug-in	Eclipse Plug-in
Project	M2M Project	medini QVT	M2M Project
Execution	Compiled to VM	Interpreted	Interpreted
Transformation specification	Hybrid	Declarative	Imperative
Direction	Unidirectional	Bidirectional	Unidirectional
Cardinality	M-to-N	M-to-N	M-to-N
Trace generation	Automatic	Automatic	Automatic
Side effects	Yes	No	Yes
Unique features	Compiled transformations Refining mode	Bidirectional	Black-box operations

Table 3.1: Comparison - A comparison of the three transformation languages and engines

One thing all transformation tools discussed have in common, is that they are implemented as plug-ins for the Eclipse platform. ATL and QVTo are both part of the M2M project of the Eclipse Foundation, while medini QVT is implemented by ikv++, a commercial company.

Chapter 4

Metrics in MDE

Size and complexity are two properties that are often used to compare artifacts in software engineering. They are, however, not always clearly defined. This chapter explores the concepts of size and complexity for the artifacts found in MDE and discusses problems related to these properties. These metrics have been used to measure the properties of the artifacts used in our experiment.

4.1 Size

The relative concept of size is often used to compare software engineering artifacts. This is also the case for entities in Model-Driven Engineering.

4.1.1 General

Oxford Dictionaries [31] defines *size* as follows:

> "[...] 10.a The magnitude[1], bulk, bigness, or dimensions of anything.
> [...] 11.a. A particular magnitude or set of dimensions."

The first definition does not provide us with a lot of information, being too general for our use. The second provides us with more insights. It defines a mapping function

[1] "b.1.b In physical sense: Greatness of size or extent." [31]

which sorts dimensions into sets. Dimensions are properties of a certain item. However, unlike other properties, a name for example, dimensions cannot be used on their own; they are only of use to us when comparing two elements.

4.1.2 Language

When considering languages, we note that their size can be measured in two ways: the number of users, and the size of the language vocabulary.

If we look at programming languages, it is hard to exactly say how many programmers use a language. We can look at the number of new applications appearing in a language, or how much a language is discussed. Tiobe.com [39] maintains such a list.

When we look at the vocabulary of the languages, i.e. the number of keywords, we find that ANSI C contains 32 keywords, Java has 50, and C++ has 74 keywords. This is another way of measuring language size.

Shaw [32] aims to reduce the compilation cost of a program through language contraction. The author defines the size of a language according to the size of the compiler needed to compile a program in that language:

> "Unfortunately, the notion of 'language size' or 'language richness' is not well quantified. However, it is reasonable that compilers for 'larger' languages tend to be larger than compilers for 'smaller' languages. Therefore, compiler size is used as a crude measure for language size."

4.1.3 Model

Booch et al. [7] consider a model diagram too large if it spans more than one page. This may be a suitable measure when dealing with printed documentation, but not so much when focusing on digital models as used in MDE.

Lange [23] defines more relevant metrics, looking at number of classes, use cases or sequence diagrams. Another way of measuring the size of a model is to consider the size of the XMI file in which the model is serialized. Using this as a metric, however, we are bound to one tool and XMI version, since different tools and versions may generate different outputs for the same diagram. Another measure mentioned is the ratio between the number of diagram elements and the number of classes: $\frac{NumberOfElements}{NumberOfClasses}$.

Heijstek [13] also measures model size according to the number of classes in the model or diagram. The author also considers the number of attributes per class to be of importance.

According to Weil and Neczwid [45], it was agreed upon during the 2006 Model Size Metrics Workshop that counting model elements is a relevant metric. However, due to different levels of model abstraction, these metrics might not allow for direct comparison between models. A solution that was suggested is to define metrics in a metamodel and measure the models accordingly.

Monperrus et al. [26] argue how the number of elements can be successfully used to define model size. The authors heavily rely on the traditional way of counting software

size. Since not all metrics are applicable in every situation, the authors recommend that the Goal/Question/Metric (GQM) approach [33] is to be used to define specific metrics.

4.1.4 Program

Traditionally, most programs are measured using the Lines Of Code (LOC). Grady and Caswell [11] define a LOC as a non-commented source statement. By excluding blank lines and lines of comments, only the real application code is counted. [10] gives us two example calculations that use LOC:

1. Let NCLOC be the number of non-commented lines of code and CLOC the number of commented lines of code. The total length (LOC) of a program is thus NCLOC + CLOC.
2. The calculation $\frac{CLOC}{LOC}$ gives us the density of comments in a program.

However, one might argue how useful or even correct these metrics are. Programs that are created using source code generators might be verbose, using a large amount of lines. However, these might not be completely relevant, since the programmer did not write these lines himself.

Halstead [12] was one of the first to attempt to define metrics that go beyond counting lines of code. In his work, he defines (for a given program P) four metric tokens:

μ_1 The number of unique operators
μ_2 The number of unique operands
N_1 The total occurrences of operators
N_2 The total occurrences of operands

By using these metrics, we can define other metrics, like the length of P, which is calculated as $N = N_1 + N_2$ and the vocabulary of P calculated as $\mu = \mu_1 + \mu_2$. With these, we can define the volume V of P, which is to be described as "the number of mental comparisons needed to write a program of length N" [10]:

$$V = N \times \log_2 \mu$$

The program level can then be written as $L = V^*/V$ (V^* is the volume of the minimal size implementation of P), resulting in a program difficulty that is calculated as

$$D = 1/L$$

Halstead's theory goes on to explain that we can calculate the estimated program level as follows:

$$\hat{L} = \frac{1}{D} = \frac{2}{\mu_1} \times \frac{\mu_2}{N_2}$$

As we can see here, Halstead directly relates code size and difficulty of understanding the programming code, implying that a larger program in terms of lines of code results in a more difficult program to understand and maintain and vice versa. [10] discusses that this implication is arguable, citing sources that directly contradict Halstead.

Another way of measuring code size and complexity is proposed by [6]. The aim of the Constructive Cost Model is to predict software development and maintenance costs, and effort. Although the main aim is on software project management, and not on measuring software as such, it does provide us with several indications that program size and complexity are directly correlated with the effort it requires to alter or add functionality to the application. For example, the Intermediate COCOMO formula has the following form:

$$E = a_i \times (KLoC)^{b_i} \times EAF$$

In this formula, E is the required effort in person-months, a_i and b_i are variables that depend on the structure of the development team (organic, semi-detached or embedded) and EAF is an Effort Adjustment Factor that is calculated through a series of project, personnel, hardware and product attributes. Even though this seems like a suitable way for measuring effort, this formula is, according to [8] and [15], not correct. The first debates that more people on a project, i.e. less person-months per person, will increase the total amount of months a project will take[1], where the second argues that using (K)LOC as a metric is outdated. Due to the wide spread use of higher-level programming languages since the first usage of LOC as a metric, the non-coding activities have become more expensive than coding activities. As such, LOC may not be suitable for measuring software. Instead, *function points*, or FPs are identified as an alternative. These calculate the cost of a unit of work based on past project experiences.

However, since FPs measure software functionality based on external inputs, -outputs, -inquiries, -files and internal files, they do not provide an actual program size, rather, they tell us something about its workings. [15] identifies three types of software by using FPs:

Small	Software of less than 1000 function points.
Medium	Software that is between 1000 and 10,000 function points
Large	Software build up of more than 10,000 function points

Yet, FPs are based on more than measurable metrics, such as developer and team experience. Because of this, they are a suitable measure for software size and complexity with regard to the environment of the software development process, but not as a pure analytical tool for measuring internal software attributes.

4.1.5 Transformation

Transformations are in many ways similar to programs. As such, van Amstel et al. [41] that the use of LOC to measure model transformations has the same problems as using LOC as a metric for program size. They propose *number of functions*, *number of signatures* and *number of equations* as suitable metrics for ASF+SDF transformations.

Based on the quality attributes specified in the previous work, Vignaga [44] defines several size metrics with relation to transformations specified in ATL. For specifying

[1] This rule is known as Brook's law [8].

these metrics, the author uses the ATL metamodel as a guideline, defining metrics on properties of the elements in this metamodel. In total, 81 metrics are specified, covering both size and complexity of ATL transformation rules. Size metrics, among others, include:

- Average Size of Source Metamodels
- Average Size of Source Patterns
- Number of Rules per Source Element

The author notes that the set of 81 metrics is probably too large, indicating that some metrics might be too fine grained. When comparing these metrics to those defined in [41], the author notes that the difference in the number of metrics can be explained by the difference in size and complexity of the ATL metamodel in relation to its ASF+SDF counterpart.

4.2 Complexity

Like size, complexity is a measure often used to compare software engineering entities. Similar to size, complexity is also often ill-defined.

4.2.1 General

Complex is defined by [31] as:

> "[...] 2.a Consisting of parts or elements not simply co-ordinated, but some of them involved in various degrees of subordination."

Therefore, in order for something to be complex, it must contain multiple elements. Below we noticed that metrics often measure complexity by looking at a certain quantity of elements, and often perform calculations based on the way they are combined.

The definition as stated above is correct in all situations. However, we may add extra information when we are dealing with software engineering elements. Fenton and Pfleeger [10] define complexity in four different ways: problem complexity, algorithmic complexity, structural complexity and cognitive complexity. The first, also called computational complexity, describes the complexity of the problem that is solved by the algorithm or program. The second looks at the complexity of the algorithm. [10] notes that "this type of complexity measures the efficiency of the software". The third kind aims to quantify the complexity of the structure of the software implementing the algorithm. The final measures how much effort is required in order to understand the software.

For our purposes, we regard these concepts differently. Since we are interested in metrics that measure internal properties, we will disregard the cognitive complexity as mentioned in [10], since this takes the "human factor" into account. Furthermore, we can structure the complexity viewpoints differently, seeing them as problem complexity

and solution complexity. In turn, solution complexity encompasses algorithmic and structural complexity.

We can notice a hierarchy within these views on complexity. At the lowest level, we have a *problem*, which is solved by an *algorithm*, which is implemented in *software*, which is to be understood by the *programmer*.

4.2.2 Language

A natural language is often seen as complex when the grammar of said language is extensive. Artificial languages can be categorized using the Chomsky hierarchy [35]:

Type 0 Recursively enumerable languages
Type 1 Context-sensitive languages
Type 2 Context-free languages
Type 3 Regular languages

As we go deeper in this hierarchy, more restrictions are applied to the language. Languages of type 3 are highly restricted and only include simple languages. Type 2 languages allow for more expression, including all languages of type 3 and more. Programming languages are of this type. Languages of type 1 are rarely used in computer science. An example of a context-sensitive grammar is the syntax of a natural language, in which the context determines whether a word is accepted by the grammar. Type 0 languages include all of the language types mentioned above. These languages are characterized by the possibility of being verified by a Turing machine in a finite amount of time.

4.2.3 Model

Monperrus et al. [26] define a measure of *structural complexity* of models as follows:

$$SC = \frac{|A|}{|M|}$$

Here, the number of edges, or relations, is denoted by $|A|$ and the number of modules, or classes, by $|M|$. The result is, according to the authors, the degree to which understandability of a system is affected.

Chidamber and Kemerer [9] take a broader approach and define metrics for object-oriented software design as a whole. However, some of these metrics can also be applied to models. The *Depth of Inheritance Tree* (DIT) metric gives us insight of the position of an element in the inheritance tree. The idea behind this metric is that an element deeper in the tree inherits more methods and properties. Deeper trees are also considered more complex in terms of design. Finally, an element deeper in the tree reuses more code than others do. The metric is defined accordingly:

$$\mu(C) = n$$

Where $\mu(C)$ is the DIT of element C and n the depth in the inheritance tree. When an element inherits from more than one element, the value of $\mu(C)$ is counted by using the deepest tree.

The *Number of Children* (NOC) metric is related to the DIT metric. As the name implies, NOC is determined by the number of direct subordinate elements. This metric was chosen due to the ability of children to reuse properties from their parents. In addition, a large number of children could increase the likelihood of improper abstraction. Finally, many children could cause problems when changing the parent element, since this would require additional testing. Like DIT, the definition of NOC is straightforward: the value is solely determined by one factor, namely the number of children:

$$\delta(C) = m$$

The third relevant metric is *Coupling Between Object Classes* (CBO). This metric calculates the number of other classes to which a class is coupled. We use the term classes, rather than objects. CBO is relevant for maintainability, since a high degree of coupling reduces modularity and prevents reuse. Due to tight design, changes elsewhere in the system could have a large impact on highly coupled classes. As with the other two metrics discussed before, a high degree of coupling could mean that additional testing is required.

Heijstek [13] also defines coupling metrics:

Dep_Out	Number of elements on which the element depends.
Dep_In	Number of elements depending on the element.
NumAssEl_scc	Number of associated elements in the same namespace.
NumAssEl_sb	Number of associated elements in the same scope branch.
NumAssEl_nsb	The number of associated elements not in the same scope branch.
EC_Attr	Number of times the element is externally used as an attribute type.
IC_Attr	Number of attributes in the element having another element as their type.
EC_Par	Number of times the element is externally used as parameter type.

All these metrics can help define the degree of coupling of model elements. This is relevant for model quality, since high coupling often indicates low modularity, low degree of reusability and extra effort needed when testing.

4.2.4 Program

Developed by McCabe in 1976, the *cyclomatic complexity number* [24] is one of the best known measures of software complexity. It denotes the number of linearly independent paths through a program. The author represents an application as a directed graph. Using graph theory, he defines complexity as follows:

$$v(G) = e - n + p$$

Here, $v(G)$ is the cyclomatic number of graph G, with n vertices, e edges and p connected components. He defines a node as a block where the flow is sequential; branches in the program are denoted by arcs in the graph. Several properties hold true for the cyclomatic complexity number [24]:

1. $v(G) \geq 1$.
2. $v(G)$ is the maximum number of linearly independent paths in G. It thus is the size of a basis set.
3. Inserting or deleting functional statements to G does not affect $v(G)$.
4. G has only one path if and only if $v(G) = 1$.
5. Inserting a new edge in G increases $v(G)$ by one.
6. $v(G)$ depends only on the decision structure of G.

[10] notes that $v(G)$ on its own is not an intuitive measure of complexity, yet it does provide us with a good indicator of the difficulty of testing and maintaining a program or module.

Woodfield [46] directly relates the complexity of the problem that is to be solved with the complexity of the software solution, noting that a 25% increase in problem complexity results in a 100% increase of solution complexity. However, the author assumes that the effort required by the programmer is directly related to the program complexity. As we have noted earlier, this is not always the case, since the effort required by an experienced programmer will be less than that of a novice programmer.

4.2.5 Transformation

[41] defines two categories for transformation metrics: *function metrics* and *module metrics*. The first encompass, among others, the *number of signatures per function*, the *number of equations per function*, the number of values a function takes as an argument and the number of values it returns. These last metrics are called *val-in* and *val-out*. *Fan-in* and *fan-out* are also named as suitable metrics, indicating the number of callers and callees of a function respectively.

Module metrics are concerned with the module as a whole, rather than parts of it. Metrics at this level are, for example, number of library modules, number of times module m is imported by other modules and number of imported declarations in module m. These metrics are thus concerned with the degree of coupling between modules, even though the authors do not name this as such. Again, fan-in and fan-out are named by the authors.

According to Vignaga [44], a number of complexity metrics can be borrowed from the realm of object oriented programming. Examples for transformations are:

- Number of Attributes
- Average Height of Hierarchies of rules
- Average Number of Variables per rule
- Cyclomatic Complexity of a Helper
- Fan-In of Libraries
- Fan-Out of Libraries

4.3 Problems

Problems caused by an increase of size and complexity are encountered by software engineers and users alike. Below, we identify problems caused by the growth of these two internal properties.

4.3.1 Language

Not only a large language can cause issues, but languages that are not large enough may be troublesome too [32]: if language features are omitted to reduce language size, overhead may occur due to inefficiency when writing programs. Languages that are too large may cause the developer to either ignore parts of the language, or become confused by the language possibilities. Programs written in large languages take longer to compile than programs written in subsets of said language.

4.3.2 Model

Complex models may imply that more testing is needed on the modeled software. This is not a problem for the model as such, but it is a problem for the artifacts generated from the model. Furthermore, large or complex models are harder to grasp when working with them. An experienced software engineer might be able to understand these complex models, yet a novice engineer might have difficulties with the same task. Herein lies one of the difficulties with defining model size problems: most metrics are geared at the development process, and the degree of expertise of the user is disregarded. As a result, certain metrics might define a model as simple, yet it may be hard to understand for the user. It is therefore not possible to describe problems in terms of difficult and simple.

A problem that can objectively be measured, is that beyond certain sizes models can no longer fit in computer memory, which may cause the application working on the model, or the computer as a whole to stop working. We were not able to find sources that provide exact numbers, since model size is highly dependent on the modeling tool used and the version of the modeling language. Computer configurations, both hardware and software, also play a role in this problem. As such, it is not possible to give exact numbers with regard to this issue.

4.3.3 Program

Stobie [34] argues that the size and complexity of modern software systems causes problems with regard to testability, since traditional handcrafted tests are not sufficient anymore. He also comments on several misconceptions surrounding testing. One of the examples mentioned is that code coverage in testing determines the quality of the test. Due to software size, this is often not possible in a reasonable amount of time, thus causing non-testers to believe the testing was not performed properly. According to the author, solutions to these problems can be found by providing high quality unit testing, static checking and concurrency testing.

Another problem of programs becoming too big, is that they no longer fit in the memory of the computer. This is often a problem with model checking, where it is caused by state space explosion: the exponential growth of the number of program states the model checker has to verify. This can cause the model checker to become too large for the computer memory.

Jones [15] identifies several problems with regard to program size at a code level.

Abeyant defects	Some defects occur at runtime, but are impossible to reproduce due to a unique combination of system state and hardware and software. This may cause the error to occur once in the lifetime of an application.
Externally caused defects	If a software system becomes large and encompasses different domains, defects may be caused due to external changes in, for example, laws or government mandates. Although not present at the time the program was created, domain changes may invalidate assumptions made or facts known during the initial development.
Bad fixes	Patches created for a broken module may cause other parts of the system to become broken. Systems that are larger tend to be more prone to these problems.
Reused defects	Ideally, systems reuse code in order to prevent duplication. However, if a reusable module is broken, this may cause problems throughout the whole system. If the fix for a certain module is a bad fix, additional problems may occur.
Defect potentials	If a system grows in new development iterations, it becomes more defect prone: more code means more places defects can occur. If we combine this with the previously mentioned problems of testing large systems, we can see that it is harder to keep a large system defect-free than smaller systems.

4.3.4 Transformation

According to [41], size is related to the degree in which a model transformation can be understood or altered[1]. If a transformation becomes larger, these quality attributes decrease. The authors further claim that the size of the domain-specific part of the transformation is more important than the size of the domain-independent part. Since the domain-specific part is specific to a single problem domain, it is unlikely to be reused in other problems. In contrast, it is noted that a large domain-independent part is suitable for reuse, as they provide a library of functions to be used in other solutions.

The size of functions is thought of as having a negative effect on understandability and modifiability. When looking at the val-in and val-out metrics, the authors of

[1] These are two external attributes that cannot be measured objectively.

[41] conclude that high values for these metrics indicate highly specialized functions, resulting in poor reusability. The opposite holds true for high values of fan-in and fan-out, since these metrics may indicate that a function is used as a library function, resulting in better reuse.

When looking at modules, the results are quite similar to the metrics defined for functions. A high number of library functions is beneficial for reuse, whereas a high number for the fan-in and fan-out metrics might benefit reuse, but reduces understandability.

During the Transformation Tool Contest (TTC) events [2], solutions to problems with model transformations are submitted and compared. The problems discussed are related to various quality properties. The topics include expressivity, evolvability, performance, scalability and the ability of tools to solve certain problems. Although the Tool Contest gives an insight in the current strengths and weaknesses of transformation tools, a clear focus of achieving comparable results, statistical soundness and metrics over models and transformations is lacking.

4.4 Conclusion

Size is defined as a set of dimensions. However, the elements of this set are not always well-defined. Since sources contradict or complement each other on what to measure in certain artifacts, we can conclude that the metric for the size of an artifact depends on the reason why we are measuring the artifact. Most metric values ought to tell something about understandability, however, this term is hard to quantify and heavily relies on the user. In order to measure software artifacts objectively, countable elements should be used, which must then be counted using well defined metrics.

Complexity is a concept which is defined in an abstract way. Even more than in the case of size, definitions of complexity are related to understandability by the human user. Objective complexity metrics often entail calculations on size metrics, leaving the idea that without size, complexity cannot be evaluated. Complexity of languages is ill-defined, only allowing them to be structured using the Chomsky hierarchy. Model complexity metrics are usually derived from the area of graph theory. This is also the case for programs, which are translated into graphs, after which the metrics are calculated. Transformations can be treated in the same fashion.

Problems due to size or complexity can be caused by elements being either too large or too small. The former is well known, often being associated with low understandability and usability. The latter is equally important, as can be seen with languages: if a language is too small, the developer is limited in his expresiveness. If a language is too large, developers are given too much choice.

Models that are deemed too big or too complex either can not be used correctly by the user, or by the computer in which the model is manipulated. The user might not be able to grasp the model, or the model might be too big for the computer to load or save. Models are rarely deemed too small, with metamodels being the exception, since these are used to define a language. Programs that become too large or complex,

suffer the same problems as models, including some additional ones regarding their runtime character. The size and complexity problems of transformations are similar to those of programs. The difference is that transformations often include a domain-independent and a domain-specific part, where size and complexity in the latter is of greater importance, since it is the least likely to be reused.

Metrics are often introduced as a "one-stop solution" for problems with regard to artifact size and complexity. However, due to their nature, metrics need to be chosen based on the requirements of artifacts. The GQM approach may be used in order to aid these decisions.

Chapter 5

Transformation experiments

In this chapter, we discuss our experiments which will be used to investigate the performance of the three model transformation languages. The two chosen model transformations shall be explained and justified. A performance comparison between ATL, QVTo and QVTr is made and the alternative implementation strategies ATL_a *and* ATL_b *are evaluated.*

5.1 Model transformations

A model transformation relies on several factors. The first entity we can identify is the *transformation definition*, which contains the description of the transformation that is to be performed. Secondly, we have the *source model*. This model is an instance of the *source metamodel*, which is the third element of our transformation. The *target model* and *target metamodel* are the fourth and fifth respectively.

When performing the transformation, the transformation engine takes these five elements and performs computations based on them. The result is a mapping from the source model to the target model, which adhere to the source and target metamodels respectively. This mapping is based on the transformation definition.

5.2 Experiments

The experiments that were conducted are two-fold: firstly, a comparison of the implementation of the transformation engines of the three model transformation languages discussed in Chapter 3 is made. Secondly, the performance based on different implementation strategies of one transformation has been compared.

5.2.1 Comparison of Engines

To compare the implementation of the transformation engines, we implemented two transformations in each language. Firstly, the SimpleClass2SimpleRDBMS transformation was implemented. In this transformation, a simplified version of the Unified Modeling Language class diagram is used. The result is a database schema, which represents the class diagram in a tabular structure, including foreign keys as relations between tables.

This transformation had already been implemented for all three transformation languages. Only the name of the transformation definition varies between languages. Although all transformations achieve the same result, they differ in their approach. As such, none of them accepts the same model and metamodels, and the algorithms to achieve the mapping are different. The transformations were altered such that the transformations became algorithmically identical and accepted the same input, resulting in transformations where the only difference is the language in which they are written.

The second transformation chosen was the RSS2ATOM transformation. This choice was made to see in what way the structure of the input model affected the performance of the model transformation. The SimpleClass2SimpleRDBMS input models have a graph-like structure, whereas the RSS2ATOM input resembles a tree.

The RSS2ATOM transformation takes a Really Simple Syndication file as an input and transforms it into Atom syndication file. Both these standards are used for the publication of web-content that can be automatically processed. Examples of these include news items and blog posts.

The mayor difference between the two standards is that ATOM allows any arbitrary payloads to be sent with the message, whereas RSS restricts this to plain text or HTML. However, this is not of an issue in our experiments: only plain text was used as payload, since we were interested in the capabilities of the transformation languages, not in the differences between the two standards.

Since this transformation was only available in ATL, the QVTr and QVTo implementations were written with the algorithm of the ATL implementation in mind.

5.2.2 Comparison of Implementations

Besides the performance of the different transformation engine implementations, we were interested in the performance of different language constructs. To study the effects of alternative implementations, we reimplemented the SimpleClass2SimpleRDMS transformation using two different strategies:

Move navigation to attribute helpers (Transformation $\mathbf{ATL_a}$) The initial ATL implementation of the SimpleClass2SimpleRDBMS transformation uses navigation expressions over the input model in the right-hand side of the bindings in the matched rules. Since these navigations are potentially reused, these expressions were moved to attribute helpers. This caused the transformation to execute the navigation once, after which the result is cached for later reuse.

Implement imperatively (Transformation $\mathbf{ATL_b}$) Instead of using a declarative notation for the transformation definition, the algorithm was rewritten to be executed imperatively.

5.2.3 Transformation input

To test the SimpleClass2SimpleRDBMS transformations with regard to both size and complexity, the input models were generated with a different number of classes, an indication of size, and number of attributes per class, as an indication of complexity. The second assumption can be made due to the fact that all attributes of a class are of a different type, resulting in high CBO values. The models were generated using Epsilon. Epsilon is part of the Generative Modeling Technologies (GMT) project of The Eclipse Foundation [38]. The code used to generate these models can be found in Appendix B.1 on page 59.

The generated models contained one through one thousand classes and a number of attributes in the same range. As a result, a class may contain one through one thousand attributes, the model containing one through one thousand classes in total. Each class in the same model had the same amount of attributes. There was, however, one exception, namely the model containing a single class. Here, it was not possible to let the class contain more than one attribute, since each attribute in a class had to be of a different type[1].

The models were generated for two scenarios. The first scenario was a general usage case, in which not all model elements needed to be transformed. The results of this test illustrate how the transformation tools perform in general. The second set of models exemplified a worst case scenario. Here, all model elements needed to be evaluated and transformed to the output models. These results would show us the slowest performance of the tools that could be achieved with comparable input models.

Unlike the SimpleClass2SimpleRDBMS transformation, it was not possible to change the input model along both size and complexity axis for the RSS2ATOM transformation: the hierarchy of the RSS metamodel is quite flat, without relations between the different elements of the model. As such, the input model was only enlarged, rather than made larger and more complex. Again, the input models were generated using Epsilon.

[1] This was defined in the generation rules.

5.3 Execution

In order to compare the three transformation languages and their implementations, the transformations mentioned above were executed with the same input metamodel and the same set of models: one set of models was generated for all languages, as to prevent differences in execution times being caused by differences in the input.

Before executing the transformations, metrics have been gathered belonging to the different MDE entities. This was done using the tools described in [40] and [42] for the ATL and QVTo transformation languages respectively. [20] describes a tool for measuring QVTr metrics, however, this tool was not publicly available at the time of writing. As such, no metrics for QVTr have been extracted. While executing, all execution times were recorded. These times included the time required to load the metamodels, models, transformation definition and the serialization of the results.

The experiments were conducted on a computer system running the 64 bit edition of Microsoft Windows 7 Professional. It was equipped with a quad core Intel Core i7 920 CPU, each core running at 2.67 GHz. The system had 6 GBs of RAM. The Eclipse Platform version used was 3.5, Galileo[1]; Eclipse ran on the 32 bit version of Java 1.6, build 16, with HotSpot version 14.2. Due to the nature of Eclipse, it ran on a single CPU core, using a maximum of 1 GB of RAM.

The model transformation code can be found on [1].

5.4 Transformation explanations

To obtain a good comparison between the three transformation languages, all transformations should accept the same input metamodels and model. This has been described in the previous sections. However, in order to perform the measurements of the transformations, some alterations were made to the transformation tools.

Firstly, the ATL and QVTo plug-ins did not mention the correct times required by the transformation or did not present us with an execution time at all respectively. The ATL Eclipse plug-in allowed us to print the time taken by the execution, however, the times required to load the metamodels and models, and to serialize the result were not included here. QVTo did not provide us with any information regarding execution time at all. Through alterations to the source code of both plug-ins, we were able to retrieve the times needed for the full transformation. No alterations were made that could affect the execution times.

Secondly, the medini QVT tool did print execution information, however, this information includes all matched rules and the result of transformations per model element. When performing the transformation using models that included over fifty elements, this resulted in `Exceptions` to be thrown by the Eclipse environment. Again, through alterations in the source code of plug-in, these verbose messages where reduced

[1]The QVTr transformations were performed using Eclipse 3.4, since this was the version used by medini QVT

to the same amount of output generated by the other two plug-ins, allowing medini QVT to transform larger models too. The time required to execute the transformation includes the loading and serializing of metamodels and models, so no alteration was needed here.

Thirdly, next to outputting a lot of debug information, medini QVT is the only tool that automatically serializes the traces of the execution. Not taking very long for smaller input models, this became a problem when executing transformations on large models, requiring a lot of overhead for this serialization step. This feature was thus removed from the plug-in.

Finally, when attempting to execute the ATL_b implementation, we found that the ATL Virtual Machine has an internal stack of one hundred elements. Since the imperative implementation pushed intermediate results on this stack, but they were not popped from it immediately, ATL_b could not be executed using model with more than one hundred elements as an input.

5.5 Explanation on metrics

In Chapter 4, we have discussed metrics that can be used when dealing with MDE artifacts. Since many metrics were collected, not all are used to measure the models and transformations. Twelve metrics were selected to measure the metamodels and models, twenty-five metrics are calculated for the ATL transformations, and seventeen are extracted from the QVTo transformations. The tool required to measure the metrics for the QVTr transformation was not available at the time of writing [20]. Some side-notes have to be made with regard to the selected metrics.

Firstly, since *enumerations* are not equal to full classes, they are not counted as such. Although they appear in a similar way in class diagrams, they cannot contain variables and methods.

Secondly, when looking at the list of model metrics, we see a pattern of metrics counting the minimum, maximum and average of some properties. Due to the nature of the *Minimum Number of Children* and the *Minimum Depth of Inheritance Tree*, the values of these are always zero; this is the case for leaf- and root nodes respectively. We have chosen not to include these metrics in the result tables.

Lastly, no concrete metrics have been calculated for the input models. Since the metrics described are calculated by hand, and the input models contain vast numbers of classes, attributes, relations, or RSS items, it can be seen that manual calculation for these models can be regarded as not feasible in a reasonable amount of time.

5.6 SimpleClass2SimpleRDBMS

5.6.1 Performance

Figure 5.1 and Figure 5.2 contain the resulting times obtained from performing the SimpleClass2SimpleRDBMS transformations. The first contains the times needed by the transformations to map an increasing number of classes, with the number of attributes per class fixed at one hundred. The second illustrates the opposite, depicting the times needed to transform input models with an increasing number of attributes, the number of classes fixed at one hundred. Both figures illustrate the results of the general case scenario and the worst case scenario. When comparing the graphs of the different language implementations we can note several things.

Firstly, when comparing the graphs in Figure 5.1(a), we find that when dealing with an increasing number of classes, ATL and ATL_a perform nearly identically, both having slow slopes. On the other hand, QVTr has a steep slope, indicating that a high number of classes has a big effect on the performance of the medini QVT tool in a general case scenario. QVTo has a performance between these two languages. Due to some problems with the ATL_b implementation, only three data-points are plotted, which can hardly be seen because of the other lines.

When looking at the worst case input results in Figure 5.1(b), we observe a bigger difference between the transformations and tools. We can also see a small difference between the initial ATL implementation, and the implementation with the navigation moved to attributes. What is not apparent from this figure, but can be observed in Table A.14 on page 56, is the fact that the QVTr implementation has become the fastest performing transformation.

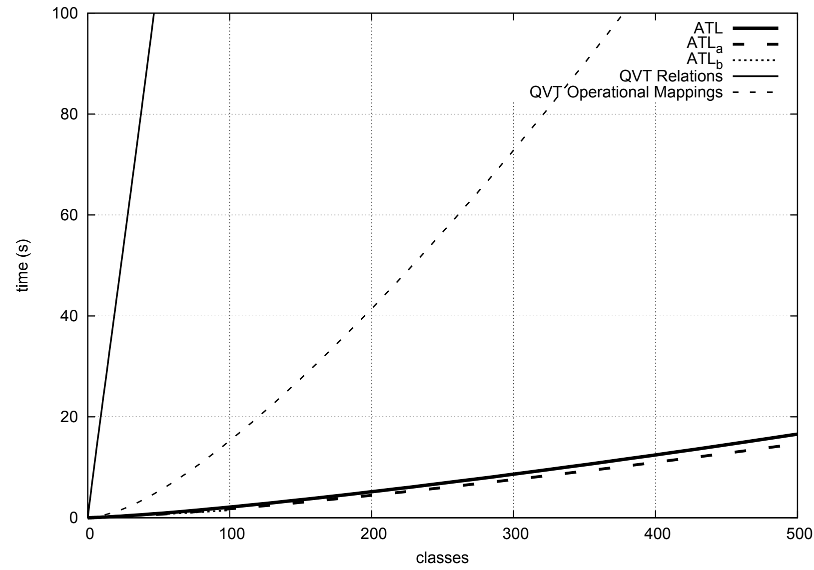
If we look at the second pair of figures, we again see differences with regard to the previous graphs. In Figure 5.2(a), we notice that the initial ATL transformation is the best performing implementation, quickly followed by ATL_b[1]. The graph of ATL_a illustrates that in this case the optimization done has an adverse effect. QVTo performs in a similar fashion to this ATL implementation. QVTr is, like in the experiment, the slowest performer.

When comparing these results with the results from the worst case scenario input set, we see that QVTr, unlike in Figure 5.1, now performs identically to Figure 5.2(a). QVTo has become slower, leaving the two ATL transformations as the fastest two. Here, the initial transformation performs slower than the optimized one.

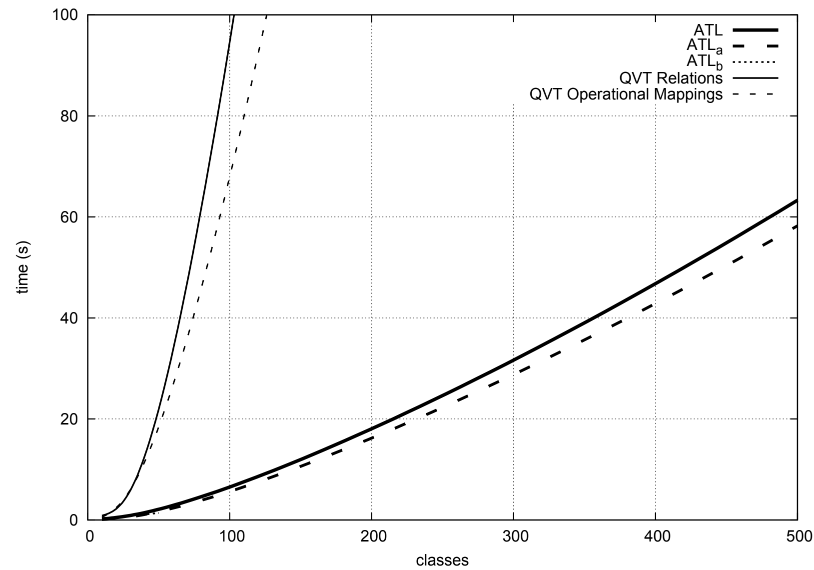
5.6.2 Metrics

The metrics extracted from the ATL transformations can be found in Table A.3 on page 53. What we find here, is that all three transformations exist of five transformation rules. Of these rules, five are matched rules in the initial transformation and in ATL_a.

[1]This is the only graph that shows the results from this implementation, Appendix A.2.1 should be consulted for further results.



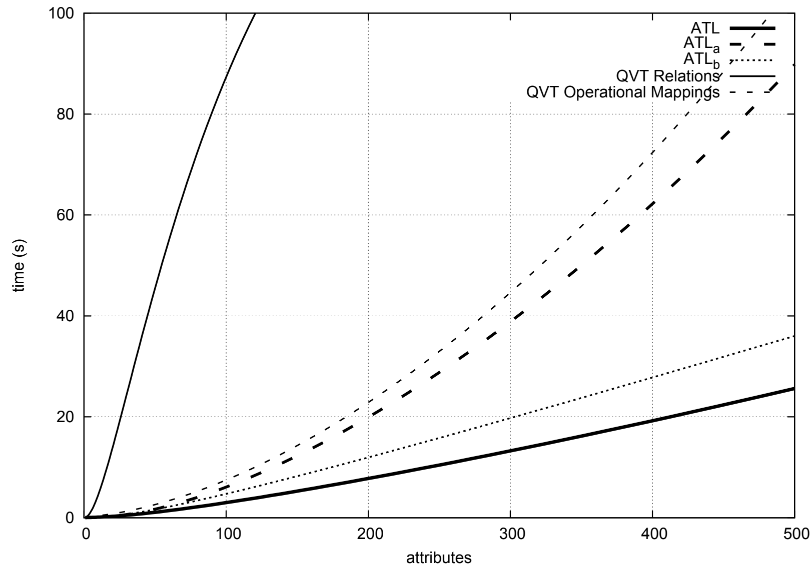
(a) General case



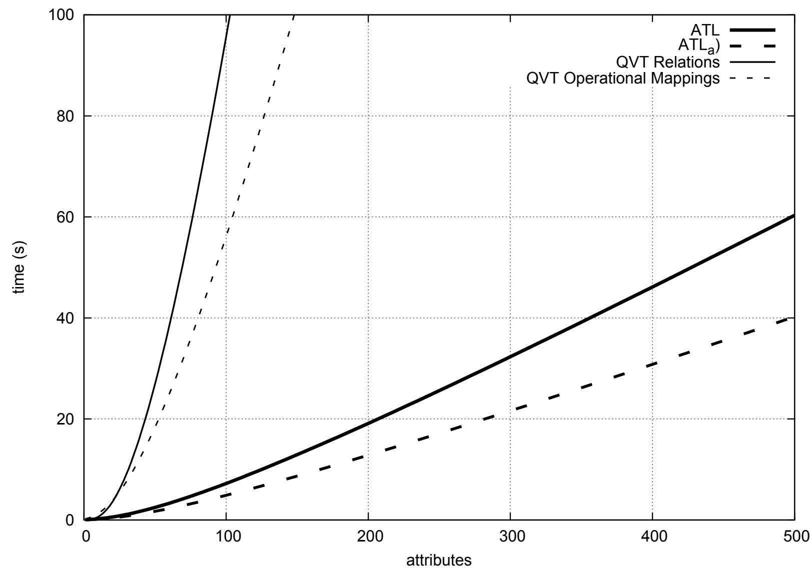
(b) Worst case

Figure 5.1: SimpleClass2SimpleRDBMS - Comparison between transformation tools, varying number of classes

ATL_b is the only implementation that requires called rules. In ATL, target model elements can only be generated by rules. Imperative transformations need called rules to generate these target model elements. This type of rule has a `do`-section to build the



(a) General case



(b) Worst case

Figure 5.2: SimpleClass2SimpleRDBMS - Comparison between transformation tools, varying number of attributes

desired target elements in an imperative way. On average ATL_b contains one statement per `do`-section. This implies that most `do`-sections are only used to generate a target model element, and are not used to perform additional calculations.

All three transformations contain at least one helper, and one or more of these helpers is an attribute helper. ATL_a contains only attribute helpers, since input model navigation is moved here. A difference between the original ATL transformation and the ATL_a implementation, is the number of times `allInstances()` is called. The metrics suggest that these calls have been moved from rules in the initial implementation to helpers in ATL_a. In ATL, this method is used to navigate the source model. Since the navigation is moved to attribute helpers, this shift in metrics was expected.

When looking at the fan-in metric for helpers, we see that the average value for this metric is higher for the declarative version of the transformation than for the implementation where the navigation is moved. This is caused by an increase in the number of helpers for the latter version.

When looking at the QVTo transformation, we see from the metrics that this transformation is nearly the same size as the ATL implementations. The number of helpers is low, only one is specified. This was expected, since the algorithm is identical to the initial ATL transformation, using the same language constructs. The fan-in and fan-out values are low, as a result of matching done by the transformation engine, instead of explicit calling.

5.7 RSS2ATOM

5.7.1 Performance

In the graph of the RSS2ATOM transformation results, we see the same curves as in the SimpleClass2SimpleRDBMS results: QVTr is quickly outperformed by the other two. When comparing ATL and QVTo, we find that the QVTo transformation takes approximately twice the time to complete than is required by ATL. However, the times measured are merely 0.500 seconds apart, so the significance of this difference is arguable.

5.7.2 Metrics

Unlike the ATL implementations for the SimpleClass2SimpleRDBMS transformation, the ATL version of the RSS2ATOM transformation uses more non-lazy matched rules and no helpers. Since the rules have on average more bindings, they are more complex. However, when looking at the standard deviation of this metric, we see that this number is high, indicating that this high average of the complexity of rules is caused by a small number of rules. The average fan-in is also lower than for the SimpleClass2SimpleRDBMS implementations. This is caused by the higher number of non-lazy matched rules that cannot be invoked explicitly. These rules are instead matched by the transformation engine. As a result, we see less explicit navigation in the rules, causing the number of calls to `allInstances()` to be lower in this transformation definition.

Like in the SimpleClass2SimpleRDBMS transformation, the metric values for the QVTo implementation of the RSS2ATOM case are nearly the same due to the algorithms being identical.

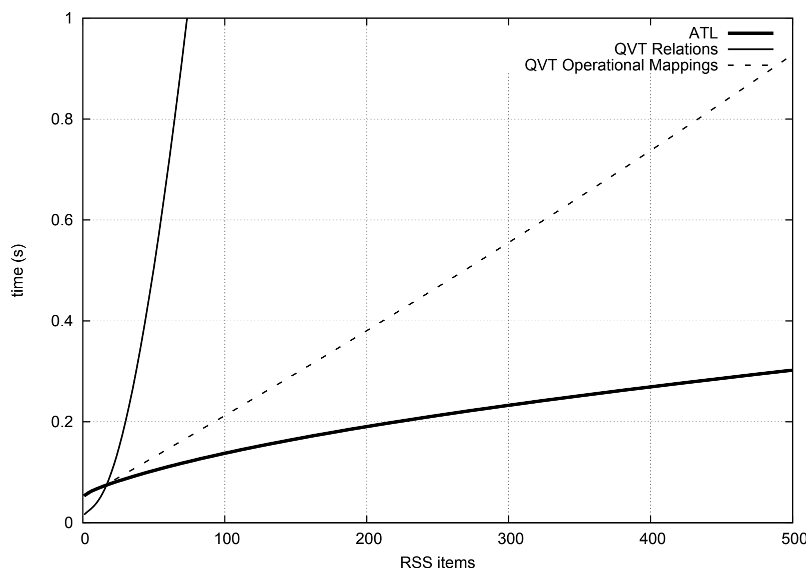


Figure 5.3: RSS2ATOM - Transformation languages compared using the RSS2ATOM transformation

5.8 Analysis

To gain additional insights into the influence of the different implementation strategies of the SimpleClass2SimpleRDBMS transformation, a statistical analysis has been performed [43]. The authors related the metrics obtained from the model transformation definitions to the execution times measured. Using linear regression, it was estimated how the execution times of the transformations depend on the metric values.

5.8.1 Moving navigation to attributes

The main difference between the initial ATL implementation, and the implementation in which the navigation of the input models had been moved to attributes, is the number of attribute helpers. As such, this metric has been used for our regression analysis. Our results indicate that there exists a negative relation between the number of attribute helpers and the execution time, meaning that the use of attributes is beneficial for the performance of the transformation execution. However, results show a large standard error on the coefficient, meaning that they are not statistically significant. A possible explanation for this is that the number of subjects used for the analysis is not enough; only two implementations are used in our experiments. This may be solved by creating hybrid transformations, where half the navigation is moved to attributes.

5.8.2 Declarative vs. imperative

When considering the differences between the declarative and the imperative implementations of the SimpleClass2SimpleRDBMS transformation, we see that the latter uses called rules; the former does not use this language construct. Metrics indicate that the imperative implementation contains four called rules, therefore, we used this metric for our regression analysis. The results show a positive relation between the number of called rules and the execution time, meaning that the performance degrades as more called rules are used, i.e. when it is opted to use an imperative implementation. Unfortunately, as is the case with the previous analysis, these results are not significant due to a lack of data. Again, this may be solved by creating hybrid transformations, this time implementing half of the transformation imperatively, while keeping the rest declarative.

5.9 Discussion

5.9.1 Comparison of languages

Overall, the ATL implementation is the best performing. When we increase the size or the complexity[1] of the input models, the transformations run over five times faster than their QVTo and QVTr counterparts in a general usage scenario. This difference grows when we are dealing with a worst case scenario. Here, QVTo and QVTr have similar performances. QVTr performs better in a worst case scenario, then when dealing with our general case models. During the testing of the former, the medini QVT tool encountered `HeapSpace` problems and became unresponsive when input models had more than 100 classes or 100 attributes. These problems did not arise when inputting the worst case models. Due to our limited knowledge of the implementation of the QVTr transformation engine in medini QVT, we cannot explain this behavior.

ATL, like QVT, was also affected by this change in input models, however, to a lesser extent.

5.9.2 Comparison of language constructs

If we compare the transformation languages, we see that the ATL transformation engine implementation performs the best out of the three. If the size or complexity of the input models is altered, the relative performance difference between the languages remains the same. However, when we compare different language constructs within a single language, we see that these also have an effect on the performance.

When comparing the results, we see that if we increase the number of attributes in our general case scenario, ATL_a becomes slower than the initial ATL transformation. This can be seen in Figure 5.1(a). This suggests that caching may not always be beneficial. The reason for this performance change ought to be sought in the input

[1] In these experiments, the number of attributes is solely used to measure complexity.

models. If the input model does not contain elements that are visited more than once, caching may have an adverse effect on the performance. However, if the elements are queried more often, caching will have a positive effect, since the results needed are calculated only once. Considering the metrics, we cannot claim that a declarative transformation with more attribute helpers will perform better than a transformation that uses navigation in the bindings. Helpers should be defined for classes that are often matched by rules and often queried.

The imperative implementation $\mathrm{ATL_b}$ is slower than its declarative counterparts when the size of the models increases. However, we were unable to obtain results for all the input models due to a stack size problem. If this problem is resolved, we will be able to run the imperative transformation for more models in order to obtain more reliable results.

In general, we may conclude that declarative transformations perform better in ATL than imperative ones. We can indicate several metrics that determine the "declarativeness" of an ATL transformation definition. The metrics that count the number of matched rules, the number of called rules, and the number of `do`-sections give use a good overview of the degree to which a transformation is either declarative or imperative. As a rule of thumb, we can say that the bigger the number of matched rules, the more declarative the transformation is. On the contrary, greater values for the called rules and `do`-sections metrics indicate an imperative implementation style.

5.10 Conclusion

For our transformation performance experiments, we choose two distinct transformations: a transformation working on graph-like input models, and a transformation that would map between two tree structures.

Using these two, the performance of ATL, QVTo, and QVTr were measured. Different implementation strategies were taken for the ATL transformation, resulting in two alternate implementations: $\mathrm{ATL_a}$, where the navigation over the input model was moved to attribute helpers, and $\mathrm{ATL_b}$, which was implemented imperatively.

After repairing some initial inconveniences due to a discrepancy between our requirements and the features the tools offered, the transformation experiments could be executed successfully. It was found that ATL is the best performing language overall. Based on our results, we can say that QVTr performs the slowest. When looking at different language constructs, we found that using attribute helpers may benefit the performance of the model transformations in ATL. Implementing transformations imperatively may cause a decrease in performance.

Chapter 6

Conclusion

In the previous chapters, we have researched metrics to measure size and complexity in Model-Driven Engineering, and have measured and compared their effects on the performance of model transformations and model transformation tools. In this chapter, we will give our conclusions and recommendations for future work.

6.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a software development process that focuses on models, as we have seen in Chapter 2. In MDE, everything is considered a model, which is an instance of a metamodel. A metamodel is a description of a modeling language. An MDE process considers languages, models, programs and transformations as development artifacts.

When dealing with models, we distinct three levels, each with a higher level of detail: metametamodels, metamodels and models. Metametamodels define the language of metamodels, which in turn define the language of a model.

Since all software artifacts can be considered as models, we can use model transformations to translate between them. A model transformation uses a definition, which has two metamodels and one instance of these metamodels as an input. The result is a model that is an instance of the other metamodel. Transformations can be used to either translate between languages[1] or refine a model within the same language.

[1] As defined in the metamodels.

6.2 Transformation languages and tools

In Chapter 3, we listed three model transformation languages and tools that are currently the most popular in the field of MDE.

Firstly, the ATLAS Transformation Language is a hybrid transformation language that allows developers to write their model transformations both declaratively and imperatively. By using the refining mode, transformations can be written to refine the source model, a feature that is not present in the other two transformation languages. Instead of creating a new model, the source model can be altered to include new information.

To work with ATL, we used the Eclipse M2M ATL plug-in. Unique about this tool when compared to the other two language tools is its compilation feature, which compiles the ATL code to an intermediate language.

Secondly, the Query/View/Transformation Relations language is a purely declarative transformation language. It allows developers to write bidirectional transformations in a declarative fashion. QVTr is implemented in medini QVT. This tool is an interpreter for QVTr transformations, consisting of a parser, analyzer, evaluator and a visitor.

Thirdly, the QVTo language is an imperative transformation language. Unlike QVTr, QVTo transformations are unidirectional. Transformations can benefit from external libraries by using black-box operations. These allow developers to access code written in other languages and use it in their transformations. The tool used for working with QVTo is the Eclipse M2M QVTo plug-in, which operates in the same fashion as the interpreter used in medini QVT.

6.3 Metrics in MDE

In Chapter 4, we looked at size and complexity metrics for MDE artifacts. Size was defined as a set of dimensions, but the exact definition of these dimensions was not always given for an artifact. When the dimensions are defined, they may not always be appropriate for the situation at hand. As such, we can conclude that these metrics should not be taken as a given, rather they should be used in their appropriate situation. The GQM approach may be used for the selection of the right metrics.

In our research, we were interested in objective measures. The metrics we found included many that required information about the user of the software artifacts, resulting in a human factor that had to be taken into account. The same is true for complexity. It is not possible for us to define “too complex” without knowledge of the environment in which the artifact is to be used. As such, metrics requiring information about the user are unsuitable for our purposes of measuring artifacts objectively.

Problems that result from an increase in size or complexity are on either an application or a comprehension level: when models become too large, they may not fit in the computer memory. Another problem might be that the developer no longer understands them. When models become too complex, this problem might arise too. This is also the case for programs and transformations.

6.4 Transformation experiment

Chapter 5 discussed our performance experiment. Two transformations were analyzed: SimpleClass2SimpleRDBMS and RSS2ATOM. These were chosen because of their different structures; the first has a graph-like structure, the second is structured like a tree.

After running the model transformations, we found that the ATL implementation was the fastest, followed by QVTo and QVTr respectively. The implementation of the latter ran into `HeapSpaceException`s when inputting models with more than 100 classes or 100 attributes in the general case scenario. This problem was not encountered when using the worst case scenario models. This behavior could not be explained due to our limited knowledge of the tool implementation.

By comparing three different implementations of the ATL SimpleClass2SimpleRDBMS transformation, we found that the caching provided by attribute helpers may be beneficial to the performance of model transformations, if the input models contain elements that are frequently queried. If this language construct is used improperly, the performance may be degraded. Generally, the declarative transformations performed better than the imperative transformation. This, however, could not fully be tested due to problems with the ATL Virtual Machine.

6.5 Evaluation

In Chapter 1, we described our objectives to this research. We defined size and complexity in the domain of MDE in Chapter 4. The metrics discussed here were used to analyze the performance of model transformation tools. These tools have been studied and compared in Chapter 3, as per our second objective. Their performance has been analyzed in Chapter 5, which concludes that the ATL implementation of the Eclipse M2M project is the best performing tool of the three tools discussed.

Furthermore, through the use of alternative implementations ATL_a and ATL_b, we found that the use of attribute helpers in ATL may benefit the performance of model transformations, whereas the use of an imperative implementation strategy would decrease the speed at which a model transformation definition could be executed.

We can conclude that the overall objective, an investigation of the performance of model transformation languages and an evaluation of the effect of size and complexity of transformation elements has been fulfilled.

6.6 Future work

The research discussed in this thesis is not complete, rather, it is a starting point for additional research in the field of performance measures for model transformations and tools. Based on our experiences, we can make the following recommendations for future work:

Statistical significance Through the research performed in this thesis, we can make several assumptions with relation to model transformation language implementations and the use of model transformation language features. However, these assumptions cannot be proved to be statistically significant due to a lack of data. This work should be continued by evaluating a greater diversity of model transformation definitions in order to provide additional performance information.

Real world use cases We have used two small examples for our research to establish a baseline analysis of the performance of the model transformation tools. Real world use cases should be evaluated in order to gain insight into the performance of the model transformation tools in practice.

Additional model transformation implementations When testing the performance of alternative transformation implementations, a selection was made with regard to the language features that were used. As such, not all language constructs have been evaluated. This is the case for both ATL, where we did not use inheritance, and QVTo, where queries could have been used to separate the navigation from the mappings. Future work should evaluate the effects of alternative language constructs on the performance of model transformations.

Additional model transformation languages and tools In Chapter 3 we discussed three languages and their features. These three, were then used to perform the model transformation performance analysis. Although these three languages and implementations are the most popular transformation languages at the moment, more languages and language implementations exist. In order to gain extensive knowledge on the overall performance of model transformations, these additional languages and tools need to be covered.

The research discussed in this thesis is valid for the versions of the tools as described in Chapter 3. Newer versions of the transformation engines could perform differently.

Compiled vs. interpreted transformation engines A difference in language implementation we observed was that one transformation engine was implemented as a compiler (ATL), whereas the two others were implemented as interpreters (QVTo and QVTr). A comparison between these two implementation strategies would be interesting.

References

[1] "Performance experiments with ATL and QVT." [Online]. Available: http://wwwhome.cs.utwente.nl/~kurtev/ATLQVT/

[2] "Transformation Tools Contest." [Online]. Available: http://is.tm.tue.nl/staff/pvgorp/events/TTC2011/CfC.pdf

[3] F. Allilaire and T. Idrissi, "ADT: Eclipse development tools for ATL," Université de Nantes, LINA, Tech. Rep., 2004. [Online]. Available: http://atlanmod.emn.fr/www/papers/ADT_AllilaireIdrissi.pdf

[4] *Specification of the ATL Virtual Machine*, ATLAS group, 2005.

[5] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui, "First experiments with the ATL model transformation language: Transforming XSLT into XQuery," in *OOPSLA 2003 Workshop*, Anaheim, California, 2003. [Online]. Available: http://www.softmetaware.com/oopsla2003/bezivin.pdf

[6] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, W. A. Brown, S. Chulani, and C. Abts, *Software Cost Estimation with Cocomo II*, Hardcover, 2000. [Online]. Available: http://www.worldcat.org/isbn/0130266922

[7] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[8] F. P. Brooks, Jr., *The Mythical Man-month (anniversary ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, 1994.

[10] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. Boston, MA, USA: PWS Publishing Co., 1998. [Online]. Available: http://portal.acm.org/citation.cfm?id=580949

[11] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-wide Program*. Boston, MA, USA: Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1987. [Online]. Available: http://portal.acm.org/citation.cfm?id=580949

[12] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series).* Elsevier, 1977. [Online]. Available: http://www.worldcat.org/isbn/0444002057

[13] W. Heijstek, "Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process," in *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on 27-29 Aug. 2009*, 2009, pp. 113–120.

[14] ikv++ technologies ag, "medini qvt." [Online]. Available: http://projects.ikv.de/qvt

[15] C. Jones, *Software Engineering Best Practices.* McGraw Hill, 2010.

[16] F. Jouault and I. Kurtev, "On the architectural alignment of ATL and QVT," in *Proceedings of the 2006 ACM symposium on Applied computing, Dijon, France.* New York: ACM Press, April 2006, pp. 1188–1195.

[17] F. Jouault and J. Bézivin, "KM3: A DSL for metamodel specification," in *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, 2006.

[18] F. Jouault and I. Kurtev, "Transforming models with ATL," in *MoDELS Satallite Events*, ser. Lecture Notes in Computer Science, J. M. Bruel, Ed., vol. 3844. Springer, 2005, pp. 128–138.

[19] F. Jouault, J. Bézivin, and I. Kurtev, "TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering," in *Proceedings of the 5th international conference on Generative programming and component engineering*, ser. GPCE '06. New York, NY, USA: ACM, 2006, pp. 249–254. [Online]. Available: http://doi.acm.org/10.1145/1173706.1173744

[20] L. Kapov, T. Goldschmidt, S. Becker, and J. Henss, "Evaluating Maintainability with Code Metrics for Model-to-Model Transformations," in *Proceedings of the 6th International Conference on the Quality of Software Architectures (QoSA'10)*, ser. Lecture Notes in Computer Science, vol. 6093. Springer, 2010.

[21] S. Kent, "Model-Driven Engineering," in *Integrated Formal Methods.* Springer, 2002, pp. 286–298.

[22] I. Kurtev, "Adaptability of model transformations," Ph.D. dissertation, University of Twente, 2005.

[23] C. F. Lange, "Model size matters," *Lecture Notes in Computer Science*, vol. 4364, pp. 211–216, 2007.

[24] T. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, pp. 308–320, 1976.

[25] J. Miller and J. Mukerji, "MDA guide version 1.0.1," Object Management Group, OMG document, 2003. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01

[26] M. Monperrus, J. Champeau, and B. Hoeltzener, "Counts count," in *2nd workshop on Model Size Metrics*, 2007.

[27] Object Management Group, *Meta Object Facility (MOF) Specification*, Object Management Group OMG Adopted Specification formal/02-04-03, 2002.

[28] ——, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Final Adopted Specification ptc/05-11-01*, Object Management Group OMG document, 2005. [Online]. Available: http://www.omg.org/docs/ptc/05-11-01.pdf

[29] ——, *Meta Object Facility (MOF) Query/View/Transformation Transformation Specification*, Object Management Group Std., Rev. formal/2008-04-03, 2008.

[30] ——, *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*, Object Management Group Std. formal/2007-11-04, 2007. [Online]. Available: http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF

[31] Oxford Dictionaries, *Oxford English Dictionary*, 2nd ed., J. Simpson, Ed. Oxford Dictionaries, 1997, vol. Volume 1 – 3.

[32] M. Shaw, "Reduction of compilation costs through language contraction," *Communications of the ACM*, vol. 17, no. 5, pp. 245–250, 1974.

[33] R. Solingen and E. Berghout, "The Goal/Question/Metric Method," *A Practical Guide for Quality Improvement of Software Development. New York, McCraw-Hill Publishers*, 1999. [Online]. Available: http://www.iteva.rug.nl/gqm/GQM%20Guide%20non%20printable.pdf

[34] K. Stobie, "Too darned big to test," *Queue*, vol. 3, no. 1, pp. 30–37, 2005.

[35] T. A. Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*, 2nd ed., T. Stone, Ed. Addison Wesley, 1998.

[36] *ATL/Developer Guide*, The Eclipse Foundation. [Online]. Available: http://wiki.eclipse.org/ATL/Developer_Guide

[37] *ATL/User Guide*, The Eclipse Foundation. [Online]. Available: http://wiki.eclipse.org/ATL/User_Guide

[38] "Epsilon," The Eclipse Foundation. [Online]. Available: http://www.eclipse.org/gmt/epsilon/

[39] TIOBE, "TIOBE programming community index," April 2010. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[40] M. F. van Amstel and M. G. J. van den Brand, "Quality Assessment of ATL Model Transformations using Metrics," in *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL'10)*, Malaga, Spain, Jun. 2010.

[41] M. F. van Amstel, C. F. J. Lange, and M. G. J. van den Brand, "Metrics for analyzing the quality of model transformations," in *Quantitative Approaches in Object-Oriented Software Engineering*, ser. QAOOSE, G. Falcone, Y.-G. Guéhéneuc, C. Lange, Z. Porkoláb, and H. A. Sahraoui, Eds., no. 12, 2008, pp. 41–51.

[42] M. F. van Amstel, M. G. J. van den Brand, and P. H. Nguyen, "Metrics for model transformations," in *Proceedings of the Ninth Belgian-Netherlands Software Evolution Workshop (BENEVOL'10)*, Lille, France, Dec. 2010.

[43] M. van Amstel, S. Bosems, I. Kurtev, and L. Ferreira Pires, "Performance in model driven transformations: A comparison between ATL and QVT," in *To be published*, ser. ICMT '11, 2011.

[44] A. Vignaga, "Metrics for measuring atl model transformations," Universidad de Chile, Tech. Rep., 2009.

[45] F. Weil and A. Neczwid, "Summary of the 2006 model size metrics workshop," 2006.

[46] S. Woodfield, "An experiment on unit increase in problem complexity," *IEEE Transactions on Software Engineering*, pp. 76–79, 1979.

Appendix A

Transformation results

This appendix contains the results of the transformations, as described and discussed in Chapter 5. First the metrics for the metamodels are listed, after which the metrics for the ATL and QVTo model transformations are given.

A.1 Metrics

A.1.1 SimpleClass2SimpleRDBMS metamodel metrics

Metric	SimpleClass	SimpleRDBMS
Size		
# Classes	5	3
Min. # Attributes per Class	0	0
Max. # Attributes per Class	2	2
Avg. # Attributes per Class	1	1
Complexity		
Structural Complexity	1.6	1.7
Max. Depth of Inheritance Tree	1	0
Avg. Depth of Inheritance Tree	0.4	0
Max. # Children	2	0
Avg. # Children	0.4	0
Continue on the next page		

Metric	SimpleClass	SimpleRDBMS
Min. Coupling Between Objects	1	2
Max. Coupling Between Objects	3	2
Avg. Coupling Between Objects	2	2

Table A.1: Metamodel metrics - The metrics of the metamodels used in the SimpleClass2SimpleRDBMS transformations.

A.1.2 RSS2ATOM metamodel metrics

Metric	RSS	ATOM
Size		
# Classes	9	14
Min. # Attributes per Class	1	0
Max. # Attributes per Class	15	7
Avg. # Attributes per Class	5	2.5
Complexity		
Structural Complexity	1	1.31
Max. Depth of Inheritance Tree	0	1
Avg. Depth of Inheritance Tree	0	0.4
Max. # Children	0	4
Avg. # Children	0	0.4
Min. Coupling Between Objects	1	1
Max. Coupling Between Objects	12	10
Avg. Coupling Between Objects	4.7	4

Table A.2: Metamodel metrics - The metrics of the metamodels used in the RSS2ATOM transformations.

A.1.3 ATL transformations

	SimpleClass2SimpleRDBMS			RSS2ATOM
Metric	Declarative	ATL_a	ATL_b	
# Transformation Rules	5	5	5	8
# Matched Rules	5	5	1	8
# Non-Lazy Matched Rules	1	1	0	3
# Lazy Matched Rules	4	4	1	5
# Called Rules	0	0	4	0
# Rules with an Input Filter	2	1	0	0
# Rules with a do Section	0	0	4	0
# Helpers	1	4	3	0
# Attribute Helpers	1	4	1	0
# Operation Helpers	0	0	2	0
# Calls to `allInstances()`	1	1	2	0

Continue on the next page

Metric	SimpleClass2SimpleRDBMS			RSS2ATOM
	Declarative	ATL_a	ATL_b	
# Calls To Built-In Functions	5	5	5	1
# Elements per Output Pattern	1	1	0.8	1
# Parameters per Called Rule	0	0	0.75	0
# Statements per do Section	0	0	1	0
# Bindings per Rule	2	2	1.6	3.38
Avg. Helper Cyclomatic Complexity	1	1	1	0
# Operations on Collections per Helper	0	1	1	0
# Operations on Collections per Rule	5	3	2	1
Avg. Rule Fan-In	1.4	1.4	1.4	0.88
Avg. Helper Fan-In	3	2	1.67	0
Avg. Rule Fan-Out	2	3	2.4	0.88
Avg. Helper Fan-Out	0	0	0	0
# Calls to `allInstances()` per Rule	0.2	0	0.2	0
# Calls to `allInstances()` per Helper	0	0.25	0.25	0

Table A.3: Metrics for the ATL transformations

A.1.4 QVT Operational Mappings (QVTo) transformations

Metric	SimpleClass2SimpleRDBMS	RSS2ATOM
# Mappings	6	8
# Mappings with Condition	5	0
# Helpers	1	0
# Calls to Resolve Expressions	3	0
# Elements per Mapping	56.33	45.25
# Parameters per Mapping	0.17	0.25
# Variables per Mapping	0.17	0
# Operations on Collections per Mapping	1.67	0.13
Avg. Cyclomatic Complexity per Mapping	1	1
# Sub-objects per Helper	12	0
# Parameters per Helper	2	0
Avg. Cyclomatic Complexity per Helper	1	0
Avg. Mapping Fan-Out	1.17	1
Avg. Mapping Fan-In	1	1.25
Avg. Helper Fan-Out	0	0
Avg. Helper Fan-In	3	0
Avg. Resolve from Mapping Fan-In	0.5	0

Table A.4: Metrics for the QVTo transformations

A.2 SimpleClass2SimpleRDBMS

The results listed in the tables below are the times required by the transformation engines to complete the model transformation given the inputs as described in the tables. The tables are structured as follows:

Rows Indicate the number of classes the input model contains.
Columns Indicate the number of attributes the input model contains.

A.2.1 ATL results

Classes/Attributes	1	10	50	100	500	1000
1	0.020	-	-	-	-	-
10	0.031	0.034	0.068	0.109	1.731	3.082
50	0.038	0.068	0.235	0.566	7.992	30.082
100	0.051	0.095	0.403	1.267	20.056	63.596
500	0.213	0.571	4.279	12.279	233.302	987.054
1000	0.636	1.835	12.079	41.712	832.716	3478.662

Table A.5: Initial ATL transformation (general case) - The speed of the ATL transformation in a general case scenario.

Classes/Attributes	1	10	50	100	500	1000
1	0.043	-	-	-	-	-
10	0.057	0.050	0.144	0.199	2.790	10.437
50	0.074	0.086	0.405	1.176	22.123	80.812
100	0.064	0.143	0.980	3.038	60.318	235.154
500	0.393	1.172	12.529	44.824	1024.647	4135.119
1000	1.294	3.785	45.514	166.233	3861.724	-

Table A.6: Initial ATL transformation (worst case) - The speed of the ATL transformation in a worst case scenario.

Classes/Attributes	1	10	50	100	500	1000
1	0.019	-	-	-	-	-
10	0.014	0.018	0.037	0.057	0.451	0.578
50	0.019	0.055	0.159	0.395	3.169	10.490
100	0.028	0.123	0.300	0.844	10.226	295.539
500	0.153	0.541	3.262	10.968	186.832	791.323
1000	0.414	1.276	10.853	37.068	741.765	3097.810

Table A.7: ATL_{a} transformation (general case) - The speed of the ATL transformation in a general case scenario, navigation moved.

Classes/Attributes	1	10	50	100	500	1000
1	0.016	-	-	-	-	-
10	0.024	0.023	0.047	0.093	0.745	2.288
50	0.047	0.062	0.277	0.711	11.39	41.029
100	0.034	0.147	0.705	2.149	40.235	153.764
500	0.296	1.014	11.724	40.737	924.342	3651.858
1000	0.796	3.126	41.536	154.394	3654.323	-

Table A.8: ATL$_a$ transformation (worst case) - The speed of the ATL transformation in a worst case scenario, navigation moved.

Classes/Attributes	1	10	50	100	500	1000
1	0.026	-	-	-	-	-
10	0.062	0.063	0.084	0.147	1.809	2.980
50	0.059	0.045	0.208	0.590	9.471	33.802
100	0.049	0.125	0.502	1.495	25.342	80.135
500	-	-	-	-	-	-
1000	-	-	-	-	-	-

Table A.9: ATL$_b$ transformation (general case) - The speed of the Language transformation in a general case scenario, implemented imperatively.

Classes/Attributes	1	10	50	100	500	1000
1	0.027	-	-	-	-	-
10	0.032	0.087	0.135	0.244	3.045	11.197
50	0.075	0.104	0.491	1.381	28.072	107.251
100	-	-	-	-	-	-
500	-	-	-	-	-	-
1000	-	-	-	-	-	-

Table A.10: ATL$_b$ transformation (worst case) - The speed of the Language transformation in a general case scenario, implemented imperatively.

A.2.2 QVT Operational Mappings results

Classes/Attributes	1	10	50	100	500	1000
1	0.044	-	-	-	-	-
10	0.042	0.055	0.136	0.328	4.523	7.304
50	0.087	0.183	0.891	1.881	32.970	126.038
100	0.174	0.275	1.465	5.810	108.878	371.326
500	1.176	2.492	24.366	94.030	2269.397	10601.735
1000	4.335	8.792	91.464	399.647	-	-

Table A.11: QVTo (general case) - The speed of the QVTo transformation in a general case scenario.

Classes/Attributes	1	10	50	100	500	1000
1	0.072	-	-	-	-	-
10	0.075	0.084	0.196	0.561	8.890	32.647
50	0.149	0.216	1.839	6.017	126.619	534.853
100	0.243	0.612	5.786	20.082	487.102	1999.072
500	2.614	8.096	111.804	498.124	13266.928	-
1000	9.033	27.442	449.542	2107.695	-	-

Table A.12: QVTo (worst case) - The speed of the QVTo transformation in a worst case scenario.

A.2.3 QVT Relations results

Classes/Attributes	1	10	50	100	500	1000
1	0.016	-	-	-	-	-
10	0.031	0.036	2.038	23.993	-	-
50	0.062	0.252	11.951	100.745	-	-
100	0.109	0.655	20.672	223.611	-	-
500	0.765	11.185	-	-	-	-
1000	2.823	46.050	-	-	-	-

Table A.13: QVTr (general case) - The speed of the QVTr transformation in a general case scenario.

Classes/Attributes	1	10	50	100	500	1000
1	0	-	-	-	-	-
10	0	0.016	0.125	0.858	99.778	808.393
50	0.015	0.031	0.608	4.244	501.588	3992.250
100	0.031	0.047	1.170	8.409	945.845	-
500	0.156	0.312	6.037	1007.716		-
1000	0.437	0.702	12.293	-	-	-

Table A.14: QVTr (worst case) - The speed of the QVTr transformation in a worst case scenario.

A.3 RSS2ATOM

Number of items	ATL	QVTo	QVTr
1	0.053	0.053	0.016
10	0.081	0.064	0.047
50	0.084	0.124	0.058
100	0.222	0.238	0.218
500	0.294	0.773	9.392
1000	0.455	2.030	64.101

Table A.15: RSS2ATOM transformation - The speed of the RSS to ATOM transformations.

Appendix B

Model generation code

This appendix contains the Epsilon Object Language (EOL) code, which was used to generate the model instances used in the experiment.

B.1 SimpleClass model generation

```
1   -- Package that will contain the new classes
    var p : new Package;
    p.name := 'Package';
    -- Number of classes to be generated
    var numberOfClasses : Integer := 100;
6   -- Number of attributes per class
    var numberOfAttributes : Integer := 1;
    -- Ratio of inheritance relations
    var partParents : Real := 0.6;
    -- Calculated number of inheritance relations
11  var nrParents : Integer := (partParents * numberOfClasses).asInteger();

    /*
     * Generates <numberOfClasses> new Classes with <numberOfAttributes>
```

```
     * Attributes.
16   */

    for (i in Sequence{1..numberOfClasses}) {
        var class : Class := new Class;
        class.name := 'c' + i;
21      class.is_persistent := true; --should be done random, leave for
            now
        p.contains.add(class);

        for (j in Sequence{1..numberOfAttributes}) {
            var attribute : Attribute := new Attribute;
26          attribute.is_primary := false;
            attribute.name := 'a_' + i + '_' + j;
            class.attrs.add(attribute);
        }
    }
31
    /*
     * Generates random Assocations for <numberOfClasses>*0.5
     */

36  for (i in Sequence{1..(numberOfClasses*0.5).asInteger()}) {
        var c : Class := Class.allInstances.random();
        for (c2 in Class.allInstances.random()) {
            var ass : Association := new Association;
            ass.src := c;
41          ass.dest := c2;
            ass.name := c.name + '_to_' + c2.name;
            p.containsAssociation.add(ass);
        }
    }
46
    for (c in Class.allInstances) {
        for (at in c.attrs) {
            at.type := Class.allInstances.random();
            while (at.type = c) {
51              at.type := Class.allInstances.random();
            }
        }
    }
```

```
56  /*
     * Generate random parent-child relations
     */

    for (r in Sequence{1..nrParents}) {
61          var c1 : Class := Class.allInstances.random();
            var c2 : Class := Class.allInstances.random();
            while (causesCycle(c2, c1)) {
                    c1 := Class.allInstances.random();
                    c2 := Class.allInstances.random();
66          }
            c1.parent := c2;
    }

    /*
71   * Makes <nrParents> classes persistent
     */

    for (n in Sequence{1..nrParents}) {
            var c1 : Class := Class.allInstances.random();
76          c1.is_persistent := true;
    }

    /*
     * Makes <nrClasses>*1.5 attributes primary
81   */
    for (i in Sequence{1..(1.5*numberOfClasses).asInteger()}) {
            var a : Attribute := Attribute.allInstances.random();
            a.is_primary := true;
    }
86
    /*
     * Recursively check if adding <code>check</code> as the parent of
     * <code>origin</code> causes cyclical inheritance to occur.
     */
91
    operation causesCycle(check : Class, origin : Class) : Boolean {
            if (check = origin) {
                    return true;
            }
```

```
96          if (check <> origin.parent) {
                    if (check.parent.isUndefined()) {
                            return false;
                    }
                    else {
101                         return causesCycle(check.parent, origin);
                    }
            }
            else {
                    return true;
106         }
    }
```

B.2 RSS model generation

```
    -- RSS
    var rss : new RSS;
3   rss.version := '2.0';

    -- New channel
    var chan : new Channel;
    rss.channel = chan;
8
    -- Channel category
    var cat : new Category;
    cat.domain              := 'http://trese.cs.utwente.nl';
    cat.value               := 'MSc project';
13  cat.channel             := chan;

    -- Channel properties
    chan.title              := 'RSS Channel';
    chan.link               := 'http://domain.dot.com';
18  chan.description        := 'Automatically generated RSS channel';
    chan.language           := 'lang';
    chan.copyright          := '(c) 2010';
    chan.managingEditor     := 'Steven Bosems';
    chan.webmaster          := 's.bosems@cs.utwente.nl';
23  chan.generator          := 'Eclipse Epsilon Project';
    chan.ttl                := 0;
    chan.rating             := 'Academic';
    chan.pubDate            := '01-01-1900';
```

```
   chan.lastBuildDate    := '01-01-1900';
28 chan.category         := cat;
   chan.skipDays.add('Saturday');
   chan.skipDays.add('Sunday');

   -- Number of items to be generated
33 var numberOfItems : Integer := 50;

   /*
    * Generates <numberOfItems> new Items
    */
38
   for (i in Sequence{1..numberOfItems}) {
      var item : Item   := new Item;
      item.title        := 'item␣' + i;
      item.link         := 'http://domain.dot.com/items/' + i;
43    item.description  := 'Item␣number␣' + i + '␣of␣channel␣' + chan.title;
      item.pubDate      := '01-01-1900';
      item.author       := chan.webmaster;
      item.comments     := '';
      item.guid         := '' + i;
48
      var item_cat : Category := new Category;
      item_cat.domain   := item.link;
      item_cat.value    := chan.category.value;
      item.category     := item_cat;
53
      chan.items.add(item);
   }
```