

KIEL UNIVERSITY, KIEL, GERMANY  
DEPARTMENT OF COMPUTER SCIENCE  
SOFTWARE ENGINEERING GROUP

**Bachelor's Thesis**

# **Improving a Transformation of Java Models to KDM**

Kim Yannik Lübbe (kyl@informatik.uni-kiel.de)

Last edited: September 27, 2012

Advised by: Prof. Dr. Wilhelm Hasselbring  
M. Sc. Sören Frey



---

Hiermit versichere ich, Kim Yannik Lübbe, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt und die Arbeit in keinem anderen Prüfungsverfahren eingereicht habe.

---

Ort, Datum, Unterschrift

---

---

## Abstract

This Bachelor's Thesis describes the process of improving a given transformation to convert Java Project Models into the Knowledge Discovery Meta-model format. This transformation is written in the Atlas Transformation Language. First we will mention practical reasons for the viability of model transformation. Secondly we list the different foundations and technologies relevant for this thesis. After an initial analysis of the performance problems, we will introduce different approaches to improve a given ATL transformation, and talk about feasibility of these approaches in our specific case.

In this case, the chosen way of improving resource usage and execution time is removing unnecessary detail from the transformation when dealing with large projects, which resulted in a boost of both speed and resource-efficiency. Next we evaluate the results of our modifications, and measure the performance improvements with regards to execution time and resource usage with the help of different test cases. The performance improvement was significant, although common optimization techniques were already applied to the project beforehand.



## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Approach . . . . .	1
1.3	Goals . . . . .	2
1.4	Document Structure . . . . .	2
<b>2</b>	<b>Foundations and Relevant Technologies</b>	<b>4</b>
2.1	Reverse Engineering . . . . .	4
2.2	Software Modernization . . . . .	4
2.3	Model Transformation . . . . .	4
2.4	Relevant Technologies . . . . .	4
<b>3</b>	<b>Initial Analysis</b>	<b>9</b>
3.1	Outline . . . . .	9
3.2	Extracting the Transformation . . . . .	9
3.3	Test Runs . . . . .	9
3.4	The ATL Profiler . . . . .	11
<b>4</b>	<b>Approach</b>	<b>14</b>
4.1	Outline . . . . .	14
4.2	Possible optimizations . . . . .	14
4.3	Improvements . . . . .	18
4.4	Choosing the Right Transformation . . . . .	20
<b>5</b>	<b>Evaluation</b>	<b>24</b>
5.1	Methodology . . . . .	24
5.2	Feasibility . . . . .	24
5.3	Performance . . . . .	24
5.4	Test Cases . . . . .	24
<b>6</b>	<b>Conclusions and Future Work</b>	<b>26</b>
6.1	Conclusions . . . . .	26
6.2	Future Work . . . . .	26
	<b>References</b>	<b>28</b>
	<b>Attachments</b>	<b>30</b>

## List of Figures

1	Layers, packages, and concerns in KDM . . . . .	5
2	The CloudMIG Approach . . . . .	6
3	The Cloud Suitability and Alignment Hierarchy . . . . .	7
4	Extracting the ATL Transformation . . . . .	10
5	Profiling the ATL transformation . . . . .	11
6	Counting Calls of Single Operations . . . . .	12
7	Total Time Needed, Grouped by Operation . . . . .	12
8	Numbers of Instructions Executed, Grouped by Operation . . . . .	13
9	Top Memory Usage of Single Operations . . . . .	13
10	Example for short-circuit boolean expressions . . . . .	15
11	Execution times of <i>including</i> . . . . .	16
12	Execution times of <i>includes</i> . . . . .	17
13	Comparison of different ways of computing an opposite relationship. . . . .	17
14	Grep count of <i>includes</i> . . . . .	18
15	Grep count of different collections . . . . .	18
16	Resolving the <i>extends</i> relationship . . . . .	19
17	Choosing the Right Transformation . . . . .	22

## List of Tables

1	Execution times of original transformation . . . . .	9
2	Bodies removed from transformation . . . . .	20
3	Statements removed from transformation . . . . .	21
4	Execution times of different transformations . . . . .	25



## 1 Introduction

This section introduces the main goals and their motivation.

### 1.1 Motivation

The analysis of complex software systems is an important part of software engineering. Analyses are used to reveal errors, monitor efficiency, and to reverse engineer systems, for instance for modernization or adaption to new environments, e.g., for use in cloud computing. Here the main problems lie in the diversity of programming languages, different computer architectures, and coding styles. An abstraction from these special cases is needed to reuse methods of analysis and to adapt software to different scenarios.

### 1.2 Approach

A useful approach to this problem is using a unified model for software systems. By abstracting from language and architecture, the need to re-develop those methods for every new case is circumvented. A publicly available technology that is able to represent different software systems is the *Knowledge Discovery Meta-Model* (KDM) [16]. KDM has been developed by the *Object Management Group* [6] and enables the use of unified analyzation and optimization methods for a wide array of software. This is accomplished by providing a common base for multiple analysis tools, independent from programming languages, or coding styles.

A widely used programming language is Java. To let Java developers benefit from the available tools that operate on KDM, it has to be possible to transform Java software into models compatible with these tools. Such a transformation already exists, and is utilized by tools like *MoDisco* [7] and *CloudMIG Xpress* [11]. This available transformation is written in the *Atlas Transformation Language* [15], short ATL, and is working for small to medium size systems. But it exhibits performance problems when transforming large software systems, to the point of not terminating the process, even on powerful computer systems. This poses the question, if it is possible to optimize the given algorithm, or, if this is not the case, if ATL has to be replaced by another transformation language.

This bachelor's thesis is written as part of a larger software engineering project at the Kiel University [5]. In the course of a semester, a group of students developed a software for managing KDM-models, the KDM-Model-Manager. It includes features like importing and exporting of KDM-projects, conversion from Java and Python software to KDM, and visualization of KDM-models.

### 1.3 Goals

- G1: Identifying performance problem(s)
- G2:
  - i. Optimizing the ATL algorithm
  - ii. Modifying it to fit different scale projects
  - iii. (Optional) Rewriting it in another language
- G3: Applying suitable boundaries to the transformation to use in different scenarios

The goal of this bachelor's thesis is to fix the performance issues occurring in Cloud-MIG's transformation from Java to KDM (G1). This will be accomplished by either optimizing the ATL algorithm with regards to larger software systems, i.e. modifying it to an extent where it can process large projects without problems. The alternative approach would be replacing the given transformation in question with a model transformation written in another language (G2.iii). A model transformation language that seems suited to this kind of problem is Xtend.

To achieve these goals, we will have to analyze the transformation currently used with regards to large software systems. Secondly we will start identifying possible performance bottlenecks by applying the algorithm to a number of applicable test cases (see section 4). Furthermore, if possible, we will optimize the current ATL code and implement the optimized algorithm into a standalone library, that will be usable by other projects.

If the achieved speed optimization and resource needs of the transformation are not enough to handle large software projects, we will cut down on the level of detail from the source model that is transformed to the target. To achieve this, we will offer multiple different transformations for multiple sizes of software projects (G3).

Finally, if the mentioned optimization is not possible, we will try to find the solution in another model transformation language, and analyze if a change to this language is a sensible approach (G2.iii).

### 1.4 Document Structure

*The remainder of the thesis is structured as follows.* Section 2 outlines the foundations and technologies relevant to this thesis. Following that, Section 3 describes the extraction of the transformation from the MoDisco libraries into a standalone eclipse project. Next, it documents the first look at the transformation and lists examples of test runs with a sample software project. At the same time we evaluate the differences between running the transformation with and without the use of the ATL Profiler. Section 4 weighs possible optimization strategies against the given source code and mentions problems with these common approaches. Next we list the modifications applied to the transformation. The next Section 5 expands on the evaluation and compares the results of different

## *1 Introduction*

---

transformations applied to different test cases. After that, the final Section 6 concludes the thesis and defines starting points for future work based on this thesis.

## 2 Foundations and Relevant Technologies

In this section we will list foundations and technologies that are relevant for the thesis.

### 2.1 Reverse Engineering

Reverse engineering means analyzing a system with two main purposes:

1. Identifying the system's components and their internal relationships.
2. Creating an abstract representation of the system

[9]

### 2.2 Software Modernization

The process of software modernization involves understanding the system at hand, extracting its rules of operation, and by using current technologies and software engineering methods, creating a new system with the same set of needed features. These include for example reverse engineering and software transformation [8].

### 2.3 Model Transformation

Model transformation is a method that enables the automation of many activities like reverse engineering, application of patterns or refactoring, which are used extensively in software development and modernization. By creating an representation of the system on a higher level of abstraction and applying a set of rules known as transformation rules it is possible to convert one or more source models into one or more output models. This process is referred to as model transformation [18].

### 2.4 Relevant Technologies

#### 2.4.1 KDM

The Knowledge Discovery Metamodel's purpose is to define a unifying standard to represent software and information systems. It includes means to represent different software artifacts, such as source code, user interfaces, databases, etc. The KDM standard specifies facts, which are compiled in sets to display behavior, structure, and data of information systems. In difference to the Unified Modeling Language (UML), which generates code in a top-down manner, KDM can be used to build models from software systems and artifacts in a bottom-up manner with a higher level of abstraction. This is achieved by using reverse engineering techniques [16].

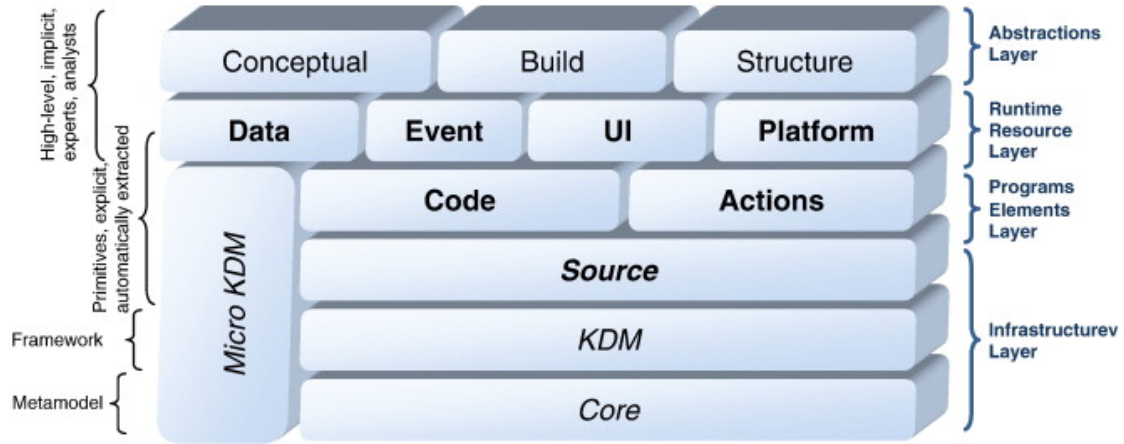


Figure 1: Layers, packages, and concerns in KDM taken from Ricardo Pérez-Castillo et. al [16]

### 2.4.2 KDM-Model-Manager

The proposed bachelor’s thesis will be written as part of a larger software engineering project at the Kiel University [5]. In the course of a semester, a group of students developed a software for managing KDM-models, the KDM-Model-Manager, including import and export of KDM-projects, conversion from Java and Python software to KDM, and visualization of KDM-models.

### 2.4.3 Java

Java is a programming language introduced by Sun Microsystems that has been increasing in popularity since 1995. Java is very popular, in fact most colleges and universities require computer software majors to take Java courses. This is mainly because of Javas simplicity, which makes it easy to learn, while being powerful enough to compete with the widely used Visual C++. Java is deployable on many platforms, because of the Java Virtual Machine, which eliminates the need for compiling against different hardware or software architectures [20].

Java is influenced by other popular languages, like C and C++. Although it is designed to solve similar problems, Java tries to eliminate the need for code complexity. Java is very familiar to users of other programming languages for these reasons. But it cuts away often repeated code and replaces it with easy to use high level functions, which makes Java different to the languages mentioned above in many areas. From this stems Javas simplicity, making it easily approachable, as it is a simple language by design, but on the other hand, Java offers more complex features like garbage collection and native

multithreading [19].

#### 2.4.4 MoDisco

Modisco is a framework to apply reverse engineering and model transformation techniques to software systems. It improves on other, existing approaches to offer a full suite of features to extract metamodels, apply model transformation rules and by that, facilitate software modernization [7].

#### 2.4.5 CloudMIG Xpress

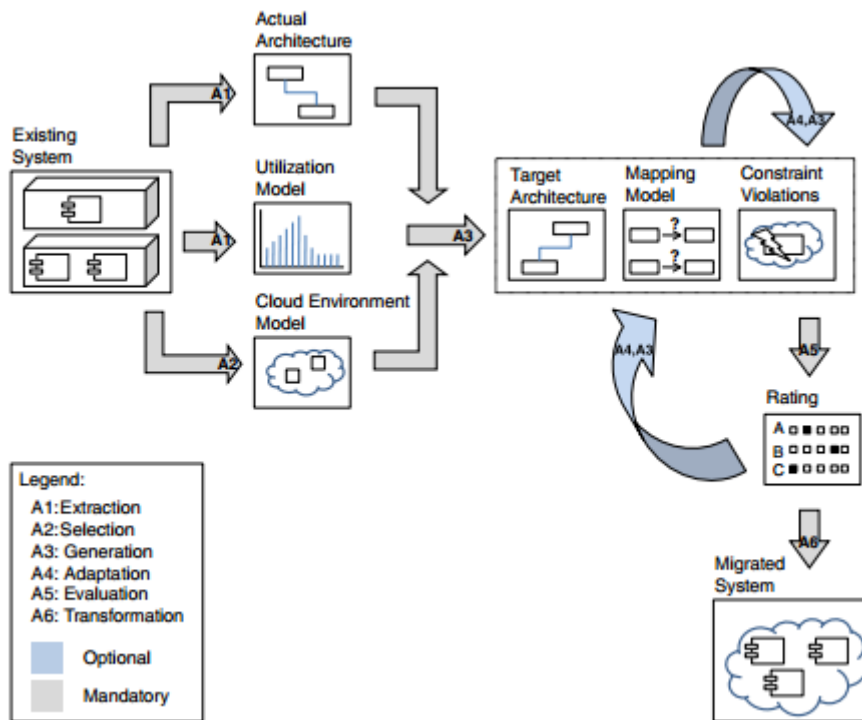


Figure 2: The CloudMIG Approach taken from Frey et al. [12]

CloudMIG Xpress is an application that provides software engineers with suitable tools to convert their software projects into a suitable format for use in cloud computing. This is achieved by making heavy use of reverse engineering techniques and model transformation to modernize software and semi-automatically adapt existing software systems to the needs of cloud based architectures [12]. An Overview of the CloudMig approach can be seen in Figure 2.

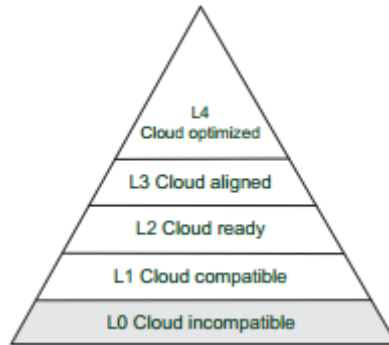


Figure 3: The Cloud Suitability and Alignment Hierarchy taken from Frey et al. [11]

CloudMIG provides means to categorize software applications into the "Cloud Suitability and Alignment (CSA) hierarchy". As seen in Figure 3, the CSA hierarchy includes categories like "cloud compatible" and "cloud optimized", to help in improving software systems and choosing the right cloud environment to deploy in. It enables software providers to semi-automatically migrate enterprise software applications to cloud based applications, with regards to resource efficiency and scalability [11].

#### 2.4.6 ATL - Atlas Transformation Language

The Atlas Transformation Language is a language for model transformation. ATL enables software engineers to convert one or more source models to a set of target models. ATL provides its own integrated development environment (IDE), which is implemented as an addon to the Eclipse platform. The ATL IDE provides users with many commonly used tools to facilitate the creating of transformation rules for use in model transformation processes, such as software modernization. These tools include, but are not limited to, an editor with syntax highlighting, and a debugger [15].

#### 2.4.7 ATL Profiler

The ATL Profiler is part of the ATL IDE. It allows the close inspection of execution count and resource usage, such as CPU time or memory, on a per-method level. The profiler dynamically analyzes the transformation rules, and outputs a detailed execution profile, e.g. a table or a tree view, which can be exported to .xmi files. Also included are sortable table and tree views, which should provide a quick way to easily detect performance issues with single or even multiple transformation rules [17].

#### **2.4.8 Xtend**

Xtend is a functional language, which could be a possible substitute for ATL. While Xtend is already suited for model transformation, the main advantage of Xtend derives from the fact that Xtend compiles into Java code, which makes it ideal for programmers already equipped with sufficient knowledge and training in Java programming. While the compiler output is easily readable, the written Xtend code tends to be shorter and more concise than the semantically equivalent Java program [14].

#### **2.4.9 Grep**

Grep, short for "global regular expression print" [13], is a UNIX/Linux command line tool that outputs all lines of a file that match a given regular expression. While powerful and flexible, grep is mainly used to find simple strings in a single file, as opposed to other search commands, who rely on matching strings or regular expressions to file names. Grep is also able to count occurrences of matches instead of printing matching lines to standard output. Matching full text strings is normally case sensitive, this can be overridden.



Transformation \ Testcase	JPetStore	Apache OFBiz	Adempiere
original	5.7508s	-	-

Table 1: Execution times of original transformation  
median of 5 runs on Intel Pentium Core i5, 6GB Java Heap

### 3 Initial Analysis

In this section, we will describe our first look at the original version of the ATL transformation.

#### 3.1 Outline

Section 3 evaluates the original transformation in two different ways. After extracting the ATL transformation into a runnable eclipse project, we try to convert our test cases mentioned in Section 5.4 from Java Project Models to KDM. Next we use the profiler program included with the ATL SDK to measure which operations from the original transformation use the most resources.

#### 3.2 Extracting the Transformation

The original ATL transformation is part of the MoDisco project. Its main purpose is to take a Java project model and convert it into the KDM format. This produces an .xmi file, which holds all the information of the original Java model, but in a format readable by the KDM-Model-Manager (See Section 2). After extracting the ATL transformation to a standalone eclipse project, as seen in Figure 4, we can now start to measure execution times using the ATL SDK.

#### 3.3 Test Runs

As you can see in Table 1, the transformation of testcase 1, JPetStore, took about 5.75 seconds on average. The other testcases, OFBiz and Adempiere, were not able to be transformed. In various tries either the garbage collector or eclipse itself crashed. Sometimes this took one to four hours, another time 3 days, but without a measurable significance or reproducible results. This was tested on computers with different system specifications and levels of hardware performance.

These test runs were all executed without the ATL profiler enabled, because the profiler produces a large overhead. As seen in Figure 5, the transformation takes nearly 5 minutes (281.8 seconds) with the profiler enabled, as opposed to about 6 seconds without the profiler. In the case of JPetStore this is tolerable, but in larger test cases as OFBiz or Adempiere, the execution of the profiler with a factor of  $\frac{281.762s}{5.7508s} = 48.995$  to the

### 3 Initial Analysis

---



Figure 4: Extracting the ATL Transformation from MoDisco to eclipse

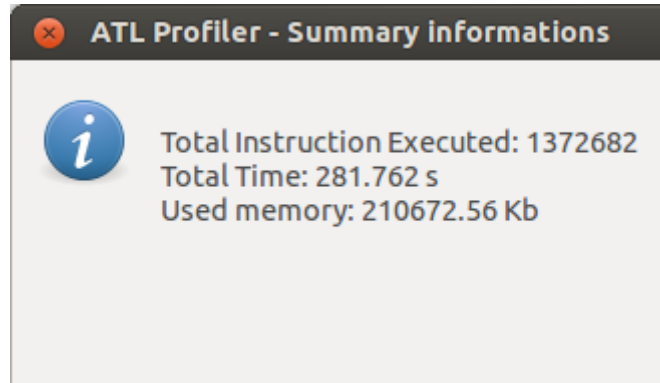


Figure 5: Profiling the ATL transformation with testcase 1 JPetStore

execution time is not practicable. This does not take into account the chance of a non-linear relation between the execution times with profiler enabled/disabled, as we lack the proper test cases for confirming polynomial or even exponential growth.

### 3.4 The ATL Profiler

The ATL Profiler has another view. It lists the different ATL operations sorted by various metrics. The interesting ones for this thesis are:

- Number of calls of this operation, Figure 6
- Total time needed for executing these operations, Figure 7
- Number of instructions executed by this operation, Figure 8
- Top memory usage of single operations, Figure 9

The evaluation of these metrics poses a problem, as either the ATL profiler or the ATL runtime seem to handle abstract methods a little different than documented on the official ATL Project website [3]. While extending an operation via abstract methods should be equal to the same static copying of the parents instructions to the child operation, abstract methods show up in the ATL profiler combined under the parent operation. This led us to replace the abstract methods with their static counterparts. While this made the source code of the original transformation harder to read, it enabled us to use the ATL profiler to its full extent. We will still refer to the static-modified transformation as the original transformation, as this is not an optimization step.











Operation Name	Calls
▶  oclIsKindOf	60883
▶  oclIsUndefined	45589
▶  getLinkBySourceElement	32138
▶  not	17830
▶  addTargetElement	11828
▶  getTargetElement	9921
▶  getTargetFromSource	9172
▶  flatten	8558
▶  addSourceElement	5221
▶  addLink2	5107

Figure 6: Counting Calls of Single Operations in the Original Transformation











Operation Name	Time Executio
▶  oclIsKindOf	102.394
▶  oclIsUndefined	58.801
▶  MethodInvocationToActionElement	41.081
▶  MethodDeclarationToMethodUnit	32.918
▶  getLinkBySourceElement	30.43
▶  AssignmentToActionElement	27.838
▶  ExpressionStatementToActionElement	19.289
▶  filterExpression	16.573
▶  FieldAccessToActionElement	15.931
▶  VariableDeclarationFragmentInFieldToStorableUnit	15.692

Figure 7: Total Time Needed, Grouped by Operation in the Original Transformation

### 3 Initial Analysis

Operation Name	Executed Instructl ▾
▶  MethodDeclarationToMethodUnit	254920
▶  MethodInvocationToActionElement	117337
▶  NamedElementToAbstractCodeElement	100918
▶  AssignmentToActionElement	92770
▶  ExpressionStatementToActionElement	83593
▶  BlockToBlockUnit	81418
▶  CreateReturnParameterUnit	68224
▶  ModifierToAttribute	65824
▶  VariableDeclarationFragmentInFieldToStorableUnit	57300
▶  ASTNodeToAbstractCodeElement	52551

Figure 8: Numbers of Instructions Executed, Grouped by Operation in the Original Transformation

Operation Name	Max Memoi ▾
▶  getTargetFromSource	258111 Kb
▶  oclIsUndefined	258111 Kb
▶  ThisExpressionToActionElement	258111 Kb
▶  getTargetElement	258111 Kb
▶  oclIsKindOf	258111 Kb
▶  getLinkBySourceElement	258111 Kb
▶  getSourceElement	258111 Kb
▶  flatten	258111 Kb
▶  getLinksByRule	249061 Kb
▶  StringLiteralToValue	249061 Kb
▶  first	249061 Kb

Figure 9: Top Memory Usage of Single Operations in the Original Transformation

## 4 Approach

This section defines how we approached improving the original transformation.

### 4.1 Outline

We start with analyzing the current ATL transformation with special regards to large software projects. Secondly, we will be identifying possible performance bottlenecks by applying the algorithm to a number of test cases, which are listed in Section 5.4. Next we will handpick redundant or useless information, that may not need to be stored by the KDM-Model-Manager, and remove these steps from the transformation. Meanwhile, we will evaluate the performance and correctness of the improved transformation by directly comparing the used resources and output to the original transformation.

### 4.2 Possible optimizations

These approaches are inspired by "Deriving OCL Optimization Patterns from Benchmarks" [10]

#### 4.2.1 Short-Circuit Boolean Expressions

To transform a model, many boolean expressions have to be evaluated in sequence. For example, in ATL, the visibility of a certain Java element can be one of 4 things:

- public
  - package (default)
  - protected
  - private
- + unknown

The fifth modifier is not part of the Java language, but must be checked for in a model translation nonetheless. We will not delve into the detailed definition of these, as it is not relevant for this thesis what these modifiers do, but how to check for them. It is only relevant that an element of the Java language, be it a class or a method, can only ever have one of these modifiers at the same time.

The easiest and most common way to check the visibility of a class, is a *switch/case* statement. In ATL, and for the sake of simplicity, a nested sequence of *if/else* statements is semantically the same. The practice of short-circuiting the evaluation of boolean expressions now puts the most common case or the most easily evaluated expression in the

```

848 -- helpers --
849
850 --returns an element of the ExportKind enumeration to set the visibility of the field or method
851 helper context java!BodyDeclaration def : getVisibility() : kdm!ExportKind =
852 | if (self.modifier.ocIsUndefined()) then
853     #unknown
854 else
855     if self.modifier.visibility.ocIsUndefined() then
856         #unknown
857     else
858         if (self.modifier.visibility = #public) then
859             #public
860         else
861             if (self.modifier.visibility = #protected) then
862                 #protected
863             else
864                 if (self.modifier.visibility = #none) then
865                     if (self.abstractTypeDeclaration.ocIsTypeOf(java!InterfaceDeclaration)) then
866                         #public
867                     else
868                         #protected
869                     endif
870                 else
871                     #private
872                 endif
873             endif
874         endif
875     endif
876 endif;

```

Figure 10: Example for short-circuit boolean expressions from the original ATL Transformation

first *if* statement. This helps to skip unwanted or redundant steps in evaluating expressions, because the first *if* statement can terminate the whole nested block of expressions and prevents other statements from executing. We see an example in Figure 10.

This is taken directly from the original transformation. As one can see, the statement is nested in such a way, that if an *if* statement holds, the rest of the block is terminated. This is desired behavior, and can not be improved on [10].

Another example of short-circuit boolean expressions is a chain of expressions connected with boolean operators, e.g. *and* and *or*. An effective way of writing *and* or *or* statement would be to make sure, the first statement is the most probable to return *true*, as this would terminate the statement and prevent other expressions from being processed. The same holds true for an *and* operator applied to a chain of boolean expressions. In this case, the optimal way would be for the first statement to return *false*, as this would terminate the whole block. This is very similar to the problem described above, as we can easily convert the list of boolean expressions into a nested *if/else* statement and vice versa.

#### 4.2.2 Collections

ATL includes 3 types of collections. These are as follows:

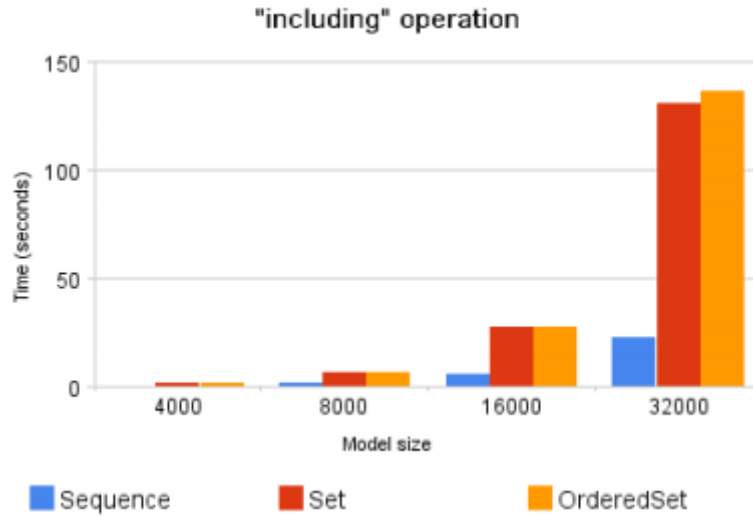


Figure 11: Execution times of *including* taken from Jesús Sánchez Cuadrado et. al [10]

- Set
- OrderedSet
- Sequence

As above, I will not get too far into the definition of these. The main point in evaluating the performance of these collection types lies in examining the operators defined for collections in ATL. The two main operators are *including* and *includes*. *Includes* checks for the existence of an element in a collection, while *including* defines an element as part of a collection, i.e. inserts the element into the collection. The three collection types in ATL behave very differently regarding to performance under these operators.

As seen in Figure 11, *sequence* is the fastest collection to use when executing *including*. While *sequence* is also the slowest when checking for elements with the *includes* method, see Figure 12, this method is never called in the original ATL transformation. To prove these and a few other points (later), we rely heavily on the usage of *grep* to find matching constructors or commands in the original transformation source code. As you can see in Figure 14, *includes* is never instantiated in the original transformation. This enables us to choose the optimal collection to use for the performance improvement of the ATL transformation as *sequence*.

As you can see in Figure 15, the collection *orderedSet* is never mentioned in the source code of the original ATL transformation *javaToKDM.atl* at all. The string *set* appears 3 times, but never as constructor to a collection, see Figure 15, lower part. The only





Figure 12: Execution times of *includes* taken from Jesús Sánchez Cuadrado et. al [10]

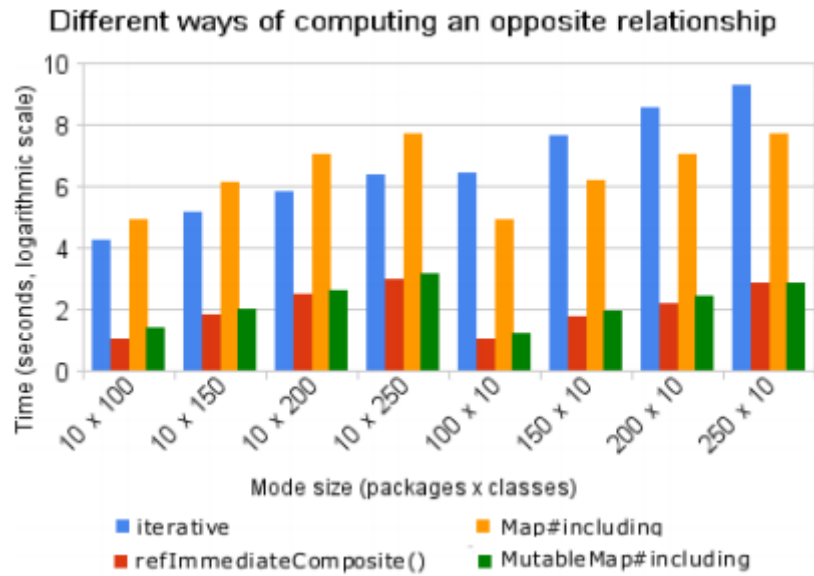


Figure 13: Comparison of different ways of computing an opposite relationship. The logarithmic scale used for the time axis corresponds to the following formula:

$$1.59 \times \ln(\text{time}) + 8.46$$

taken from Jesús Sánchez Cuadrado et. al [10]

```
$ grep -ci includes javaToKdm.atl
0
```

Figure 14: Grep count of *includes* method in original transformation

```
$ grep -ci Set javaToKdm.atl
3
$ grep -ci orderedSet javaToKdm.atl
0
$ grep -ci Sequence javaToKdm.atl
78

$ grep -i Set javaToKdm.atl
--returns an element of the ExportKind enumeration to set the visibility of the field or method
rule CatchClauseToCatchUnit extends ASTNodeToAbstractCodeElement {
rule SwitchCaseToActionElement {
```

Figure 15: Grep count of different collections in original transformation

collection ever instantiated seems to be *sequence*, with 78 mentions of the string in the source code. This is already optimal for our case, as *sequence* is the fastest usable collection in ATL [10], except for the use of *includes*, which is not used in the transformation at all.

### 4.2.3 Opposite Relationships

While transforming a source model into a target model, many child  $\leftrightarrow$  parent relationships have to be resolved. An example in our programming language in question, Java, might be the *extends* relation. The extending class in this relationship is the child, the extended would be the parent class. It is often necessary to find the opposite of the given relationship e.g. finding the parent class to a given object.

ATL provides multiple way to solve the opposite relationship problem. As you can see in Figure 13, the most efficient one is *refImmediateComposite()*. This seems like another way to look for improvable parts in the ATL transformation. The problem is, as with the other approaches, the original transformation already uses this method for resolving opposite relationships, such as *extend* or *composite*. An example for this is shown in Figure 16.

## 4.3 Improvements

This part of the thesis describes the modifications applied to the transformation. After other common optimization techniques proved not feasible in this case, we decided to lower the level of detail transformed. As mentioned in Section 3, our main problem with

```

656 --create the Extends for class or interface extension
657 lazy rule CreateExtends{
658     from
659         javaExtends:java!TypeAccess
660     to
661         kdmExtends:kdm!Extends(
662             from <- javaExtends.refImmediateComposite(),
663             to <- javaExtends->getType()
664         )
665 }

```

Figure 16: Resolving the *extends* relationship with *refImmediateComposite()* as found in the original transformation

the transformations performance was resource-usage. The level of detail on the original transformations output was fairly high, as all statements and statement bodies, e.g. loops, methods, branches like switch/case that were existent in the original project were converted and placed in the output model. Since the models are kept in XMI format, they tend to use a lot of memory while being converted. Our answer to this problem was cutting the level of detail of the output model to a more reasonable amount. For the sake of simplicity we decided on three different levels:

- full: The original output
- medium: No statement bodies in the output
- low: No statements in the output model

The results of these modifications will be evaluated in Section 5.

#### 4.3.1 Removing Statement Bodies

In ATL, statement bodies are referred to as *src.body*, and inserted into an KDM *codeElement*. By removing occurrences of these, we effectively removed the bodies of following statements:

#### 4.3.2 Removing Statements

For even larger software projects, even the removing of statement bodies is not enough, as the statements themselves account for a large portion of the source. Therefore the conversion from input to output model still used too much main memory and or did not terminate at all on the test machines. The removing of the statements themselves is our way of choice for solving this problem. For the sake of lowering the system requirements

Body removed	ATL referral
abstract methods	java!AbstractMethodDeclaration
for	java!ForStatement
foreach	java!EnhancedForStatement
while	java!WhileStatement
try	java!TryStatement
catch	java!CatchClause
synchronized	java!SynchronizedStatement
labels	java!LabeledStatement

Table 2: Bodies removed from transformation with medium level of detail

of the transformation even more, we removed following artifacts from the transformation process:

Additionally, we removed all references to the original Java source code, as this not needed for the KDM-Model-Manager. The references to the code were part of nearly every ATL operation, and as you can see in Figure 9, accounted for 3 of the seven operations using the most memory. The operations in question are:

1. *getTargetFromSource*
6. *getLinkBySourceElement*
7. *getSourceElement*

#### 4.4 Choosing the Right Transformation

The last part of this section describes our method to choose the right transformation depending on the project size. As the final transformations will be embedded into a standalone library as part of the KDM-Model-Manager project, we decided on a simple Java class to handle this choice elegantly. The source code of this class can be seen in Figure 17. The function *getProjectSize()* takes a software project file as an argument, preferably in .xmi format. After calculating the total file size it returns a custom Java Enum of type *ProjectSize*, which can be one of four values:

- *SMALL*
- *MEDIUM*
- *BIG*
- *ERROR*

Statement removed	ATL referral
Initializing arrays	java!ArrayInitializer
Access on array data	java!ArrayAccess
Access on array length	java!ArrayLengthAccess
Assignment of values	java!Assignment
Access to fields	java!FieldAccess
Access to fields of superclass	java!SuperFieldAccess
Variable Declarations	java!VariableDeclarationExpression
Conditionals	java!ConditionalExpression
Expressions in parenthesis	java!ParenthesizedExpression
Prefix operators	java!PrefixExpression
Infix operators	java!InfixExpression
Postfix operators	java!PostfixExpression
This	java!ThisExpression
InstanceOf	java!InstanceOfExpression
Null	java!NullLiteral
Booleans	java!BooleanLiteral
Numbers	java!NumberLiteral
Strings	java!StringLiteral
Types	java!TypeLiteral
Annotations	java!Annotation
Annotation members	java!AnnotationMemberValuePair

Table 3: Statements removed from transformation  
with low level of detail

```
ChooseTransformation.java ✕ ProjectSize.java ✕ *Test.java ✕
package switcher;

import java.io.File;

public class ChooseTransformation {

    // sets the maximum file size for a project to be considered small
    private static final int maxSmallProjectSizeInMB = 10;
    // sets the maximum file size for a project to be considered medium
    private static final int maxMediumProjectSizeInMB = 100;
    // everything bigger is considered a BIG project.

    public static ProjectSize getProjectSize(File projectFile) {
        if (projectFile.exists()) {
            long size = projectFile.length(); // get project filesize in Byte
            size /= 1024; // size is now in KB
            System.out.println("Size is: " + size + "KB"); // DEBUG OUT
            if (size <= maxSmallProjectSizeInMB*1024) {
                return ProjectSize.SMALL;
            } else if (size <= maxMediumProjectSizeInMB*1024) {
                return ProjectSize.MEDIUM;
            } else {
                return ProjectSize.BIG;
            }
        }
        return ProjectSize.ERROR;
    }
}
```

Figure 17: Choosing the Right Transformation depending on the Input Project

While only the first three are relevant to actually transforming the project, the `ProjectSize.ERROR` value helps to spot faulty projects or coding errors early on. This simple Java class is the interface to the rest of the KDM-Model-Manager.

## 5 Evaluation

### 5.1 Methodology

The results of this bachelors project will be evaluated by the thesis with regards to two main categories: Feasibility and performance. The main approach is to ensure an improvement in speed and memory usage, so that larger projects can be transformed without errors from Java model to KDM by the improved ATL transformation. The second goal here is to make sure that the modified transformation produces an equal output to the source model.

### 5.2 Feasibility

We have to ensure that the optimized model-to-model transformation produces an output which is equal in structure and hierarchy to the source project model. To make sure that the target model matches the source model, we did a manual review of a sample project with regards to classes and methods. We compared the target model of the original transformation to the output of the two modified transformations. The only differences of these were due to the removal of statement bodies, or statements respectively. The parts actually removed are listed in Section 4.

### 5.3 Performance

Our main goal when improving performance is decreasing memory usage and execution times. The ATL profiler bundled with the ATL SDK offers the possibility to output these execution times, see table 4. We mainly used these for performance measuring purposes. Additionally, the ATL Profiler provides means to measure the execution time of single transformation rules, which come in handy when identifying performance problems and/or improvements in a more differentiated context, and in single transformation rules.

Secondly, we measure the main memory usage of the ATL transformation, as this is a bottleneck when dealing with software projects in the KDM format. KDM files are stored in an .xmi file, and larger projects take up memory while being transformed. This escalates to the point of not terminating if the project models are too large or detailed.

### 5.4 Test Cases

For the sake of comparability, we will focus on 3 main test cases:

- JPetStore [4], a small java project which can be transformed to KDM by the original transformation without errors
- Apache OFBiz [2], an open source enterprise automation software project which will serve as our medium size project, and is not yet transformable by the original ATL transformation



## 5 Evaluation

---

Transformation \ Testcase	JPetStore	Apache OFBiz	Adempiere
full detail: original	5,7508s	-	-
medium detail: without statement bodies	3,6764s	-	-
low detail: without any statements	0,6242s	8726,6624s	45439,5640s

Table 4: Execution times of different transformations  
median of 5 runs on Intel Pentium Core i5, 6GB Java Heap

- ADempiere [1], an open source collection of ERP applications, as the biggest test-case

## 6 Conclusions and Future Work

This section concludes the main aspects of this thesis and refers to future work which could be based on its results.

### 6.1 Conclusions

Transforming software projects into the KDM format is a step into building and being able to use more unified tools for software modernization. The KDM-Model-Manager enables software engineers to convert their software projects into a form which is employable by other applications and tools, such as the CloudMIG Framework. The improvements applied to the transformation from Java Project Models to KDM help to broaden the target audience for these tools.

The thesis showed how to improve the transformation w.r.t execution time and resource usage, and accomplished the goals G1, G2.i, and G2.ii. The transformation was not rewritten in another language, thus eliminating the need for goal G2.iii. We found boundaries that suited our test cases, and implemented them with the use of a chooser library and three different transformations with differing levels of detail. This led to the reach of goal G3.

### 6.2 Future Work

This section lists a few points with the possibility to serve as base for future work.

#### 6.2.1 Fine Tuning the Level of Detail

At this moment, we have three different transformations with varying level of detail. This leads to two problems:

1. Redundant source code
2. Lack of flexibility in what will be transformed

Both of these problems could be fixed by a single transformation with dynamically adjustable levels of detail. This could go as fine grained as single selection of each artifact type that should be transformed, before the transformation is started. An example for this would be a model-2-model conversion with just packages and classes. Another example could be a very high detail transformation, but without certain statements, e.g. try/catch blocks or loops.

#### 6.2.2 Automatic Scaling According to System Power

Secondly, a problem with the current iteration of the transformation is the static nature of hard-coded boundaries with regards to software project file size. A desirable feature

to add here would be the option to determine the best boundaries for the current system specifications and project size. This would enable the library to change the boundaries for the project file size depending on the systems possible performance without too much user interaction.

### **6.2.3 Optimizing the Boundaries for Project Sizes**

The boundaries of Java class transformation chooser are derived from only three test cases, as seen in Section 4. This is far from optimal, and could easily be improved on. The more test cases we can transform on different computer systems, the more precise these boundaries can get. In combination with one or both of the above points, this would drastically improve the usability of the transformation library, and as a result, the KDM-Model-Manager.

### **6.2.4 Resolving the Profiler Overhead Problem**

There is another question unanswered in this thesis. The ATL Profiler produces a large execution time overhead as opposed to executing the transformation without the profiler enabled. The reason for this is not clear to this point, as is the nature of the relation between the execution times with or without the profiler. This could fall into the same category as the optimizing of the boundaries above, as many more test cases would be needed to calculate a formula exact enough to deal with this problem.

## References

- [1] Adempiere erp. <http://www.adempiere.com/> 2012-09-25.
- [2] The apache open for business project. <http://ofbiz.apache.org/> 2012-09-25.
- [3] Atl documentation. <http://www.eclipse.org/atl/documentation> 2012-09-27.
- [4] ibatis jpetstore. <http://sourceforge.net/projects/ibatisjpetstore/> 2012-09-25.
- [5] Masterprojekt – software engineering für parallele und verteilte systeme (ss 12). <http://se.informatik.uni-kiel.de/teaching/ss-12/projekt/> 2012-09-25.
- [6] Object management group. <http://www.omg.org/> 2012-09-25.
- [7] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. MoDisco: A Generic And Extensible Framework For Model Driven Reverse Engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10*, pages 173–174, New York, NY, USA, 2010. ACM.
- [8] Chia-Chu Chiang and C. Bayrak. Legacy software modernization. In *Systems, Man and Cybernetics, 2006. SMC '06. IEEE International Conference on*, volume 2, pages 1304 –1309, oct. 2006.
- [9] E.J. Chikofsky and II Cross, J.H. Reverse Engineering and Design Recovery: A Taxonomy. *Software, IEEE*, 7(1):13–17, jan. 1990.
- [10] Jesús Sánchez Cuadrado, Frédéric Jouault, Jesús García Molina, and Jean Bézivin. Deriving ocl optimization patterns from benchmarks. *Electronic Communications of the EASST*, 15:16, 2008.
- [11] Sören Frey and Wilhelm Hasselbring. The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4):342–353, 2011.
- [12] Sören Frey, Wilhelm Hasselbring, and Benjamin Schnoor. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software Maintenance and Evolution: Research and Practice*, 2012, 10.1002/smr.582.
- [13] Eric Goebelbecker. Using grep: Moving from dos? discover the power of this linux utility. *Linux J.*, 1995(18es), October 1995.
- [14] Arno Haase, Markus Völter, Sven Efftinge, and Bernd Kolb. *Introduction to openArchitectureWare 4.1.2*, 2007.

## References

---

- [15] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.
- [16] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532, 2011.
- [17] William Piers. Atl 3.1 – industrialization improvements. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL*, 2010.
- [18] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42 – 45, sept.-oct. 2003.
- [19] A. Van Hoff. The case for java as a programming language. *Internet Computing, IEEE*, 1(1):51 –56, jan/feb 1997.
- [20] Yun zheng Ding and Zhen Hu. To enlighten students' thinking of programming by java language. In *Multimedia Technology (ICMT), 2011 International Conference on*, pages 923 –925, july 2011.

## **Attachments**

This is a list of all attachments to this thesis.

### **Attachments on CD**

#### **ATLTransformation**

This folder includes the source code of the three different transformations.

- javaToKdm.atl - the original full detail transformation
- javaToKdm - new.atl - the medium detail transformation
- javaToKdm - final.atl - the low detail transformation

#### **ProjectSizeToTransformation**

This folder includes the source code to the Java class that chooses the transformation depending on the source project.