A QVT model transformation language represented by graph production systems

Master of Science thesis

By Ronald Nederpel

Enschede, Netherlands

September 2005

Graduation committee: dr. ir. A. Rensink
dr. ir. K.G. van den Berg
prof. dr. ir. M. Aksit

Chair: Software Engineering
Department: Computer Science
University: University of Twente
Abstract

English
Model transformation is the key in MDA, Model Driven Architecture. In MDA, there are Platform Independent Models (PIMs), which are models that are independent of any implementation languages. There are also Platform Specific Models (PSMs), which are models specified in a certain implementation language. Finally there is the programming code. The main goal of MDA is to automatically transform these three core models into each other. OMG has issued a Request For Proposals, called QVT (Queries, Views, Transformations), in order to define a language to specify transformations of models with a given source meta-model (a model of a model) to models with a given target meta-model. The source and target meta-models are fixed for each transformation definition, but not for the transformation language.

Graph transformations are about changing graphs into other graphs. The semantics of how to change one graph into another are represented by graph transformation rules. A graph transformation rule describes a change that transforms certain graphs into others. A set of graph transformation rules is usually called a graph production system.

This thesis takes the QVT proposal by IBM and partners, and defines a way to represent any arbitrary transformation definition as a graph production system. The intentions of the IBM proposal are kept as much as possible. Also a tool is made which can perform the graph transformations for any arbitrary transformation definition.

Dutch
Modeltransformaties is de kern van MDA, Model Driven Architecture. In MDA, zijn er Platform Independent Models (PIMs), modellen die onafhankelijk van implementatietaal zijn. Er zijn ook Platform Specific Models (PSMs), modellen gespecificeerd in een bepaalde implementatietaal. Ten slot is er de programmercode. Het hoofddoel van MDA is om deze drie kernmodellen automatisch in elkaar te laten transformeren. OMG, de Object Management Group, heeft een Request For Proposals uitgegeven, genaamd QVT (Queries, Views, Transformations), om een taal te definiëren om transformaties van modellen met een gegeven bron-metamodel (een model van een model) naar modellen met een gegeven doel-metamodel te specificeren. De bron- en doel-metamodellen staan vast voor elke transformatiedefinitie, maar niet voor de transformatietaal.

Graaftransformaties gaan over het veranderen van graven in ander graven. De semantiek van hoe een graaf in een andere graaf veranderd moet worden, wordt geregiseerde door graaftransformatieregels. Een graaftransformatieregel beschrijft een verandering, welke bepaalde grafen transformeert in andere grafen. Een set van graaftransformatieregels wordt een graafproductiesysteem genoemd.

Deze scriptie neemt het QVT-voorstel van IBM en partners als uitgangspunt en definieert een manier om elke willekeurige transformatiedefinitie te representeren als een graafproductiesysteem. De intenties van het IBM voorstel worden, voor zover mogelijk, in ere gehouden. Een programma, dat de graaftransformaties kan uitvoeren voor elke willekeurige transformatie definitie, is ook gemaakt.
# Table of Contents

LIST OF FIGURES ................................................................................................................................. - 6 -  
LIST OF ABBREVIATIONS ...................................................................................................................... - 8 -  

1  INTRODUCTION ....................................................................................................................................... - 9 -  
2  CONTEXT .................................................................................................................................................. - 11 -  
   2.1  MOF .................................................................................................................................................. - 11 -  
   2.2  MDA/ QVT ..................................................................................................................................... - 13 -  
       2.2.1  Introduction .......................................................................................................................... - 13 -  
       2.2.2  Structure ............................................................................................................................. - 15 -  
   2.3  ITL .................................................................................................................................................. - 16 -  
       2.3.1  “RULE” rname formals ....................................................................................................... - 16 -  
       2.3.2  (relatedRules)* ................................................................................................................... - 16 -  
       2.3.3  “FORALL” ranges ............................................................................................................... - 17 -  
       2.3.4  ( “WHERE” conjunct)? ....................................................................................................... - 17 -  
       2.3.5  “MAKE” targets ................................................................................................................ - 18 -  
       2.3.6  (”LINKING” trackingUses)? ............................................................................................ - 18 -  
       2.3.7  Factor ..................................................................................................................................... - 18 -  
   2.4  GRAPH TRANSFORMATIONS ....................................................................................................... - 19 -  
   2.5  GRAPH TRANSFORMATIONS FOR MODEL TRANSFORMATIONS ........................................ - 22 -  
   2.6  CONCLUSION ............................................................................................................................... - 23 -  
3  CLASSIFICATION OF ITL AND GRAPH TRANSFORMATION RULES ........................................... - 24 -  
   3.1  DESCRIPTION CLASSIFICATION AND ITL CLASSIFIED ...................................................... - 24 -  
       3.1.1  Transformation rules ........................................................................................................... - 25 -  
       3.1.2  Rule application scoping ................................................................................................... - 26 -  
       3.1.3  Relationship between source and target .......................................................................... - 27 -  
       3.1.4  Rule application strategy ................................................................................................ - 27 -  
       3.1.5  Rule scheduling .................................................................................................................. - 28 -  
       3.1.6  Rule organization ............................................................................................................... - 29 -  
       3.1.7  Traceability links .............................................................................................................. - 30 -  
       3.1.8  Directionality .................................................................................................................... - 30 -  
       3.1.9  Conclusion ........................................................................................................................ - 30 -  
   3.2  GRAPH TRANSFORMATIONS CLASSIFIED ............................................................................. - 30 -  
       3.2.1  Transformation rules ........................................................................................................... - 31 -  
       3.2.2  Rule application scoping ................................................................................................... - 31 -  
       3.2.3  Relationship between source and target .......................................................................... - 31 -  
       3.2.4  Rule application strategy ................................................................................................ - 32 -  
       3.2.5  Rule scheduling .................................................................................................................. - 32 -  
       3.2.6  Rule organization ............................................................................................................... - 32 -  
       3.2.7  Traceability links .............................................................................................................. - 32 -  
       3.2.8  Directionality .................................................................................................................... - 32 -  
   3.3  CONCLUSION ............................................................................................................................... - 33 -  
4  PROBLEM ANALYSIS ........................................................................................................................... - 34 -  
5  PROBLEMS ENCOUNTERED IN ITL ..................................................................................................... - 38 -  
   5.1  PROBLEMS IN THE GRAMMAR .................................................................................................. - 38 -  
       5.1.1  tRule ....................................................................................................................................... - 38 -  
       5.1.2  patternDefn .......................................................................................................................... - 41 -  
       5.1.3  linkOrder ............................................................................................................................ - 42 -  
   5.2  PROBLEMS IN THE ABSTRACT SYNTAX META-MODEL ....................................................... - 43 -  
       5.2.1  Assignment ......................................................................................................................... - 44 -  

INTRODUCTION

The project aims to perform a full analysis of the contents of the provided document. This involves understanding the structure, arguments, and information presented in each section. For the current task, the focus is on extracting and representing the content in a structured format. The analysis process involves identifying headings, subheadings, and key concepts, which are then organized into a coherent and accurate representation.
6 EXAMPLE TRANSFORMATION ............................................................ 48  
6.1 META-MODEL TO TYPE GRAPH .................................................. 48  
6.1.1 UML .................................................................................. 48  
6.1.2 Java ................................................................................. 49  
6.2 ITL RULES TO GRAPH TRANSFORMATION RULES .......... 50  
6.2.1 UML.ClassifierToJavaClass .............................................. 51  
6.2.2 UML.AttributeToJavaField ............................................... 52  
6.2.3 UML.OperationToJavaMethod ......................................... 53  
6.2.4 UML.ClassToJavaClass .................................................... 54  
6.2.5 UML.ParameterToJavaParameter .................................... 56  
6.3 SOURCE MODEL TO SOURCE INSTANCE GRAPH ................. 57  
6.4 APPLYING GRAPH TRANSFORMATION .................................. 59  
6.5 EXTRACTING TARGET INSTANCE GRAPH OUT OF COMBINED INSTANCE GRAPH .................. 61  
6.6 CONVERTING TARGET INSTANCE GRAPH TO TARGET MODEL ........................................................................ 62  
6.7 CONCLUSION ................................................................. 63  
7 GENERAL TRANSFORMATION ......................................................... 64  
7.1 CONVERTING META-MODELS TO TYPE GRAPHS .................. 64  
7.2 CONVERTING ITL TRANSFORMATION RULES TO GRAPH TRANSFORMATION RULES ................. 65  
7.2.1 “RULE” rname formals ...................................................... 65  
7.2.2 (relatedRules)* ................................................................. 65  
7.2.3 “FORALL” ranges ............................................................... 67  
7.2.4 ( “WHERE” conjunct)? ..................................................... 69  
7.2.5 “MAKE” targets ............................................................... 72  
7.2.6 (“LINKING” trackingUses)? ............................................. 73  
7.2.7 Factor ............................................................................. 74  
7.3 CONVERTING MODELS TO INSTANCE GRAPHS ...................... 76  
7.4 APPLYING GRAPH TRANSFORMATION ...................................... 76  
7.5 EXTRACTING TARGET INSTANCE GRAPHS OUT OF GENERATED COMBINED INSTANCE GRAPHS .......... 76  
7.5.1 Step 1 .............................................................................. 77  
7.5.2 Step 2 .............................................................................. 78  
7.5.3 Step 3 .............................................................................. 78  
7.6 CONVERTING TARGET INSTANCE GRAPHS TO TARGET MODELS ...................................................... 78  
7.7 CONCLUSION ................................................................. 79  
8 QVT2GRAPH TOOL ................................................................. 80  
8.1 DESIGN ............................................................................... 80  
8.2 DESCRIPTION ....................................................................... 82  
8.2.1 Step 1 .............................................................................. 82  
8.2.2 step 2 ............................................................................. 84  
8.2.3 Step 3 .............................................................................. 85  
8.2.4 Step 4 .............................................................................. 85  
8.2.5 Log window ...................................................................... 86  
8.3 EXAMPLE UML TO RELATIONAL DATABASE MODEL .......... 86  
8.4 CONCLUSION ................................................................. 87  
CONCLUSIONS AND RECOMMENDATIONS ........................................ 88  
REFERENCES .............................................................................. 90
## APPENDICES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A. APPENDIX A</strong></td>
<td>93</td>
</tr>
<tr>
<td>FIGURE 3.4</td>
<td>93</td>
</tr>
<tr>
<td>FIGURE 3.5</td>
<td>93</td>
</tr>
<tr>
<td>FIGURE 3.6</td>
<td>93</td>
</tr>
<tr>
<td>FIGURE 3.7</td>
<td>93</td>
</tr>
<tr>
<td>FIGURE 3.8</td>
<td>94</td>
</tr>
<tr>
<td>FIGURE 3.9</td>
<td>94</td>
</tr>
<tr>
<td><strong>B. APPENDIX B</strong></td>
<td>96</td>
</tr>
<tr>
<td><strong>C. APPENDIX C</strong></td>
<td>100</td>
</tr>
<tr>
<td>Meta-model to type graph</td>
<td>100</td>
</tr>
<tr>
<td>UML</td>
<td>100</td>
</tr>
<tr>
<td>RDBMS</td>
<td>101</td>
</tr>
<tr>
<td>Rules to graph transformation rules</td>
<td>103</td>
</tr>
<tr>
<td>Class2Table</td>
<td>103</td>
</tr>
<tr>
<td>Attr2Column</td>
<td>103</td>
</tr>
<tr>
<td>ForeignKeyForAssoc</td>
<td>104</td>
</tr>
<tr>
<td>Source model to instance graph</td>
<td>105</td>
</tr>
<tr>
<td>Applying graph transformation</td>
<td>106</td>
</tr>
<tr>
<td>Extracting target instance graph out of combined instance graph</td>
<td>107</td>
</tr>
<tr>
<td>Converting target instance graph to target model</td>
<td>108</td>
</tr>
</tbody>
</table>
List of Figures

FIGURE 2.1 – OVERVIEW OF FOUR LAYERS OF MODELS AND META-MODELS ........................................... - 11 -
FIGURE 2.2 – SIMPLIFIED MOF META-MODEL ....................................................................................... - 12 -
FIGURE 2.3 – MDA SOFTWARE DEVELOPMENT LIFE CYCLE .................................................................... - 13 -
FIGURE 2.4 – THE THREE MAJOR STEPS IN THE MDA DEVELOPMENT PROCESS .............................. - 14 -
FIGURE 2.5 – MDA STRUCTURE ................................................................................................................ - 15 -
FIGURE 2.6 – GRAPH OF CIRCULAR BUFFER ........................................................................................... - 20 -
FIGURE 2.7 – EXAMPLE GRAPH TRANSFORMATION RULE – GET OPERATION ON CIRCULAR BUFFER .... - 20 -
FIGURE 2.8 – MATCHING OF GET-OPERATION IN GRAPH OF CIRCULAR BUFFER .............................. - 21 -
FIGURE 2.9 – GRAPH OF CIRCULAR BUFFER AFTER APPLYING GET-OPERATION ................................. - 21 -
FIGURE 2.10 – EXAMPLE NEGATIVE CONDITION – PUT OPERATION ON CIRCULAR BUFFER ................ - 22 -
FIGURE 3.1 – FEATURE DIAGRAM OF MAJOR AREAS OF VARIATION .................................................... - 24 -
FIGURE 3.2 – LEGEND OF FEATURE DIAGRAMS ......................................................................................... - 24 -
FIGURE 3.3 – FEATURES OF TRANSFORMATION RULES ......................................................................... - 25 -
FIGURE 3.4 – FEATURES OF RULE APPLICATION SCOPING ..................................................................... - 26 -
FIGURE 3.5 – FEATURES OF THE RELATIONSHIP BETWEEN SOURCE AND TARGET ........................... - 27 -
FIGURE 3.6 – FEATURES OF RULE APPLICATION STRATEGY ................................................................. - 27 -
FIGURE 3.7 – FEATURES OF RULE SCHEDULING .................................................................................... - 28 -
FIGURE 3.8 – FEATURES OF RULE ORGANIZATION ................................................................................ - 29 -
FIGURE 3.9 – FEATURES OF TRACING ..................................................................................................... - 30 -
FIGURE 3.10 – MDA STRUCTURE ................................................................................................................ - 34 -
FIGURE 4.1 – GRAPH PRODUCTION SYSTEM STRUCTURE ......................................................................... - 34 -
FIGURE 4.2 – CONVERSION FROM MDA STRUCTURE TO GRAPH PRODUCTION SYSTEM STRUCTURE ... - 35 -
FIGURE 4.3 – DATAFLOW DIAGRAM OF CONVERSION FROM QVT TO GRAPHS .................................. - 37 -
FIGURE 5.1 – ITL ABSTRACT SYNTAX META-MODEL ............................................................................. - 43 -
FIGURE 5.2 – ASSIGNMENT PART OF ABSTRACT SYNTAX META-MODEL ............................................. - 44 -
FIGURE 5.3 – TRACKING USE PART OF ABSTRACT SYNTAX META-MODEL ........................................... - 45 -
FIGURE 5.4 – SIMPLIFIED UML META-MODEL ....................................................................................... - 48 -
FIGURE 5.5 – TYPE GRAPH OF UML ......................................................................................................... - 49 -
FIGURE 5.6 – SIMPLIFIED JAVA META-MODEL ........................................................................................ - 49 -
FIGURE 5.7 – TYPE GRAPH OF JAVA ........................................................................................................ - 50 -
FIGURE 5.8 – GRAPHRULE UMLCLASSIFIERTOJAVACLASS ............................................................... - 51 -
FIGURE 5.9 – GRAPHRULE UMLATTRIBUTETOJAVAFIELD ................................................................. - 53 -
FIGURE 5.10 – GRAPHRULE UMLOPERATIONTOJAVA METHOD ........................................................ - 54 -
FIGURE 5.11 – GRAPHRULE UMLCLASS TO JAVA CLASS ................................................................. - 55 -
FIGURE 5.12 – GRAPHRULE UMLOPERATIONTOJAVA METHOD ........................................................ - 57 -
FIGURE 5.13 – EXAMPLE UML CLASS ..................................................................................................... - 57 -
FIGURE 5.14 – EXAMPLE REPRESENTED AS INSTANCE OF THE UML META-MODEL .......................... - 58 -
FIGURE 5.15 – EXAMPLE AS INSTANCE GRAPH ....................................................................................... - 59 -
FIGURE 5.16 – COMBINED INSTANCE GRAPH .......................................................................................... - 60 -
FIGURE 5.17 – PART OF COMBINED INSTANCE GRAPH ............................................................................ - 61 -
FIGURE 5.18 – TARGET INSTANCE GRAPH .............................................................................................. - 62 -
FIGURE 5.19 – TARGET MODEL ................................................................................................................ - 63 -
FIGURE 5.20 – GRAPH TRANSFORMATION RULE PART RNAME ........................................................... - 65 -
FIGURE 5.21 – EXAMPLE META-MODEL .................................................................................................... - 66 -
FIGURE 5.22 – GRAPH TRANSFORMATION RULE 1 FOR EXAMPLE 1 .................................................. - 66 -
FIGURE 5.23 – GRAPH TRANSFORMATION RULE 2 FOR EXAMPLE 1 .................................................. - 66 -
FIGURE 5.24 – GRAPH TRANSFORMATION RULE 3 FOR EXAMPLE 1 .................................................. - 66 -
FIGURE 5.25 – GRAPH TRANSFORMATION RULE 1 FOR EXAMPLE 2 .................................................. - 67 -
FIGURE 5.26 – DELETION OF A PREVIOUS GRAPH TRANSFORMATION RULE FOR EXAMPLE 2 .......... - 67 -
FIGURE 5.27 – PART OF THE GRAPH TRANSFORMATION RULE FOR THE “FORALL RANGE” CLAUSE .... - 68 -
FIGURE 5.28 – PART OF THE GRAPH TRANSFORMATION RULE FOR THE “FORALL RANGES” CLAUSE ... - 68 -
FIGURE 5.29 – PART OF THE GRAPH TRANSFORMATION RULE FOR THE “FORALL TRACKER” CLAUSE ... - 69 -
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSTC</td>
<td>Distributed Systems Technology Centre</td>
</tr>
<tr>
<td>GROOVE</td>
<td>GRaphs for Object-Oriented VErification</td>
</tr>
<tr>
<td>ITL</td>
<td>IBM Transformation Language</td>
</tr>
<tr>
<td>LHS</td>
<td>Left Hand Side</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architecture</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>QVT</td>
<td>Queries, Views and Transformations</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>RFP</td>
<td>Request For Proposals</td>
</tr>
<tr>
<td>RHS</td>
<td>Right Hand Side</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
1 Introduction

Computers are considered powerful machines that can handle a big diversity of tasks, like calculating, word-processing, playing three-dimensional games, watching movies, listening to music and contact anyone in the whole world through the internet. It is easy to forget that a computer only recognizes zeros and ones. Computers process instructions in their native machine language, which use numeric codes to represent some basic operations like adding, subtracting and moving numbers. In the early years of the computers, programmers would have to write every program in machine language. Debugging would be done by watching many pages with zeros and ones to see where a typo error was made. Later on assembly languages were introduced which allowed the programmer to write short simple instructions in a more convenient way. During the years programming languages have evolved towards object oriented languages nowadays and resemble natural languages more and more. But programmers still have to write a lot of lines of code.

UML is a modelling language which is used to represent the design of a software program. Unfortunately so far this is only used for clarification and abstraction. It would be nice when one could automatically generate a program from a model written in UML, without having to manually program lines of code. This is one of the goals of a new research-subject called MDA, Model Driven Architecture. In MDA, there are Platform Independent Models (PIMs), which are models that are independent of any implementation languages. There are also Platform Specific Models (PSMs), which are models specified in a certain implementation language. Finally there is the code, which are the traditional lines of code written for executable programs. The models are not used as reference by a programmer to type in the code, but the main goal of MDA is to transform these three core models into each other. The biggest difference with traditional software engineering is that the created products, between the different steps, are models that can be understood by computers. [KLE03]

A meta-model is a model of a model. A model restricts the elements that can be in an instance of the model. There are different layers of models, where each model describes what the layer beneath it can contain. MOF is a standard top layer defined by OMG and is used to represent meta-models.[KLE03]

Model transformation is the key in MDA. OMG has issued a Request For Proposals, called QVT (Queries, Views, Transformations), in order to define a language to represent transformations of models with a given source meta-model to models with a given target meta-model. The source and target meta-models are fixed for each transformation definition, but not for the transformation language. The transformation language should be able to deal with any arbitrary MOF-compliant meta-model as source and target. [GAR03]

Graph transformations are about changing graphs into other graphs. The semantics of how to change one graph into another are represented by graph transformation rules. A graph transformation rule describes a change that transforms certain graphs into others. A set of graph transformation rules is usually called a graph production system. [REN05]
Grunske et al. [GRU05] propose to use graph transformations for model transformations. They state the following reasons for this:

- Graphs are a natural representation for models, because modelling languages are usually formalized by a visual abstract syntax definition.
- An established formal theory and formalisms for automatic application are provided.
- Graph transformation rules are easy to specify. Although there are no empirical studies to prove this.
- The complexity of graph transformation rules and the application formalisms can be hidden for the end user.

This thesis will take the QVT proposal by IBM and partners [DSTC04], and represent every arbitrary transformation definition as a graph production system. The input and output should be the same as for the IBM proposal, ITL called from now on, but all the processing has to be done by using graph production systems. This way, it can be seen if graph transformations are really a good way to model transformations for MDA, since there are no available implementations, which can handle arbitrary MDA model transformations by using graph production systems. The intentions of ITL will be kept as much as possible though.

Chapter 2 discusses the relevant context of MDA, QVT, MOF, graph transformations and ITL. After this, chapter 3 classifies both ITL and the graph production system and compares the features of both in order to see how well a graph production system can behave the same as ITL. This classification is based on Czarnecki [CZA03]. Czarnecki has analyzed the variability’s and commonalities of existing model transformation approaches. Using feature diagrams he represents areas of variation. Chapter 4 will then analyze the problem studied in this thesis and describes what steps need to be taken in order to solve this. Chapter 5 will describe some problems encountered when studying ITL and proposes solutions. Chapter 6 will show an example of ITL being transformed with a graph production system. After this chapter 7 will describe a general way to be able to transform any arbitrary example of ITL to a graph production system. After this Chapter 8 will describe the QVT2Graph tool, which has been created to show the working of the theory explained in this thesis. Finally conclusions will be drawn of this thesis and some recommendations are given.
2 Context
This chapter will describe some of the context of MOF, MDA, QVT, ITL (the IBM Transformation Language) and graph transformations. These are the key concepts used in this thesis and are therefore clarified in this chapter.

2.1 MOF
A meta-model is a model of a model. A model restricts the elements that can be in an instance of the model. Figure 2.1 shows four different layers of models [KLE03].

![Diagram showing four layers of models and meta-models]

At the bottom layer (layer M0), the system can be seen. This could be a UML diagram with UML objects which represent two customers, Dr Joe Nobody and Mr Mark Everyman, and an order, order number 200604.
In order to define what elements can be used in this UML model, they need to be defined. This is done in the model of the system, in layer M1. This model could define the classes Customer and Order, along with their attributes. It is also necessary to define what elements can be used to define an arbitrary UML model. This is done in layer M2, the model of a model. Concepts like Classes and Attributes are defined here for UML.

In order to define a standard way to define these models of models, OMG (the Object Management Group) defined the MOF language. This language is represented as layer M3, the model of a model of a model. [KLE03]

MOF is an extensible model driven integration framework for defining, manipulating and integrating metadata and data in a platform independent manner. [OMG03]

Figure 2.2 [KUR04] shows a simplified subset of the MOF meta-model, which shows the constructs of MOF. Primitive data types and the multiplicity of attributes and association ends have been omitted. Furthermore associations are unidirectional in this simplified MOF meta-model.

As can be seen the basic constructs consist of

- classes
- attributes
- associations between two classes

![Figure 2.2 – Simplified MOF meta-model](image)

The core constructs of MOF are equivalent to the UML language. MOF needs to be simpler and directly implemental though. [UML03]

Some of the key differences are:

- MOF only supports binary associations
- Associations in MOF cannot contain features
• Generalization, Dependency and Refinement are implemented as Associations in MOF [UML03]

MOF is used in MDA to define meta-models. Transformation definitions in MDA depend on a source meta-model and a target meta-model. Since both meta-models are written in MOF, any transformation definition can be created.

2.2 MDA/ QVT

2.2.1 Introduction

MDA, Model Driven Architecture, is a new standard in software engineering. Models are the key factor in MDA. Figure 2.3 [KLE03] shows the MDA development life cycle.

![Figure 2.3 – MDA software development life cycle](image-url)
The biggest difference with traditional software engineering is that the created products, between the different steps, are models that can be understood by computers. [KLE03]

Furthermore three important core models of MDA can be seen, namely:

- The Platform Independent Model (PIM), which is a model that is independent of any implementation languages.
- The Platform Specific Model (PSM), which is a model specified in a certain implementation language.
- The code, which are the traditional lines of code written for executable programs.

The models are not used as reference by a programmer to type in the code, but the main goal of MDA is to transform these three core models into each other, as is shown in Figure 2.4 [KLE03]. A PIM is transformed automatically by some tool into a PSM. The PSM is transformed automatically by some tool into code. [KLE03].

Defining a UML-model as a PIM and a Java-model as a PSM would enable a way to transform a UML-model into a Java-model automatically, and let Java code be generated eventually from this Java-model.

Unfortunately no well-established manner for transforming PIMs into PSMs exists. In 2002 OMG, the Object Management Group, issued a Request For Proposals (RFP) for a language in which transformation definitions could be written, a QVT language.

A QVT language defines Queries, Views and Transformations.

- A query is an expression, evaluated over a model, resulting in instances defined in the source model.
- A view is a model that is completely derived from another model, but cannot be modified without modifying the base model.
- A transformation generates a target model from a source model. [GAR03, CZA03]

Eight submissions were proposed and these should eventually lead to an OMG standard for defining model transformations.

The eight submissions were proposed by the following organizations:

1. Adaptive Ltd.
2. Alcatel, Softeam, Thales, TNI-Valiosys, et al. [ALC03]
2.2.2 Structure

In Figure 2.5 the structure of MDA, as well as the place of the QVT proposals in MDA, can be seen. \[\text{KLE03, DSTC04}\]

Meta-model A represents the meta-model of the source model A. In a transformation from UML to Java, this would be the meta-model of UML. Meta-model B represents the target meta-model. This would be the meta-model of Java in this example.

Model A represents the source model, which is an instance of meta-model A. In the example, this would be a UML-model for a certain application. Model B represents the target model, which would be a Java model in the example.

To transform from a certain model A to model B there needs to be a set of rules. These rules are represented as the “A to B transformation definition”. This transformation definition has meta-model A as its source and meta-model B as its target. In the example this transformation definition would describe how to transform any arbitrary UML model to a Java model.
Since there are many transformations possible (UML to Java, UML to C++, ERD to database, etc.), a language is needed in which transformation definitions from any arbitrary model A to any arbitrary model B can be defined. This language is represented in the figure as “Transformation Model Language”.

The QVT proposals are an attempt to define such a language.

2.3 **ITL**

One of the eight QVT proposals is made by IBM, DSTC and CBOP [DSTC04]. This thesis uses ITL to make the translation to graph transformations.

This proposal regards queries and views as specific transformations. ITL is a declarative specification and therefore does not focus on the implementation. Queries and transformations can be executed automatically. Furthermore it focuses on model-to-model transformations only and does not pay attention to model-to-text transformations. [DSTC04]

ITL has defined a grammar, which can be seen in Appendix B of this thesis. A tRule is the key construct which is used to define a transformation rule. The concrete syntax grammar of the tRule looks as follows [DSTC04]:

```
tRule:
  1. "RULE" rname formals
  2. ( relatedRules )* 
  3. "FORALL" ranges 
  4. ( "WHERE" conjunct )* 
  5. "MAKE" targets 
  6. ("LINKING" trackingUses )*   
  • SEMI; 
```

All subparts, indicated by the numbers in front of the lines, of this nonterminal are discussed now.

2.3.1 **“RULE” rname formals**

A rname defines the name of the rule and the formals define local variables introduced by this rule.

2.3.2 **(relatedRules)***

The nonterminal relatedRules leads to
```
( "extends" 
| "supersedes" 
)
```

```
rname formals ("," rname formals)* 
```
Extends
A rule may extend another rule. This way the source of a rule can be refined and additional model elements may be added to the target.

Supersedes
A rule may also supersede another rule. This way the source of a rule can be restricted, by negation of the new rule’s source.

2.3.3 “FORALL” ranges
nonterminal ranges leads to range (COMMA range)*
nonterminal range leads to mofType vname, which are both ID’s

“FORALL” moftype vname
This part is used to match elements in the source model. The moftype is matched and the variable name vname is bind to it.

2.3.4 ( “WHERE” conjunct)?
The WHERE clause restricts elements in the source model.
The following shows what nonterminal conjunct can lead to according to the grammar:

nonterminal conjunct leads to disjunct (“AND” disjunct)*
nonterminal disjunct leads to relation (“OR” relation)*
nonterminal relation leads to
  LBRACK conjunct RBRACK
  | NOT relation
  | link
  | patternUse
  | linkOrder
  | factor (ASSIGN | RELOP) factor

All these possibilities of conjuncts will be discussed in the following paragraphs.

“WHERE” disjunct “AND” disjunct “AND” disjunct “AND” disjunct
All disjuncts should be matched in the source model.

“WHERE” relation “OR” relation “OR” relation “OR” relation
At least one of these relations should be matched in the source model.

“WHERE” “NOT” relation
The relation should not appear in the source model.
"WHERE" link
It is not clear what a link is in ITL.

"WHERE" patternUse
A patternUse is a separate kind of rule which can be reused in other rules. The pattern can be used e.g. to match a set of elements in the source model. When this pattern is referred to in other rules then, these elements in the source model will have to be matched for these rules also.

"WHERE" linkOrder
A linkOrder defines the order of elements.

"WHERE" factor "ASSIGN" factor
This is used to demand that two factor’s are equal.

"WHERE" factor "RELOP" factor
This is used to demand that one factor is greater/smaller/etc. than another factor

2.3.5 "MAKE" targets
nonterminal targets leads to target (COMMA target)*
nonterminal target leads to
    | range
    | assignment
    | link
;
This is used in order to create elements in the target model.

2.3.6 ("LINKING" trackingUses)?
nonterminal trackingUses leads to trackingUse (COMMA trackingUse)*
nonterminal trackingUse leads to
    "LINKING" tname vname
    "WITH" assignment (COMMA assignment)*
;
A trackingUse creates a link between the source and target elements, used in a certain tRule. Such a trackingUse can be used again in another tRule to find the link between a source and target element, as created by this tRule.

2.3.7 Factor
nonterminal factor leads to
    constant
    | path
    | function
    | vname
A constant can be a string constant or an integer value.

A path can be used to access a feature, e.g. an attribute or an association, of a certain variable.

A function can be an operator on one or two integers, like addition. A function can also be an operation on strings, like concatenating a number of strings. Furthermore a function can be an operation on collections, like getting the first element of a collection.

This is just a variable name.

### 2.4 Graph transformations

Graph transformations are about changing graphs into other graphs. The semantics of how to change one graph into another are represented by graph transformation rules. A graph transformation rule describes a change that will transform certain graphs into others. A set of graph transformation rules is usually called a graph production system. According to Rensink, graph transformations are a very natural technique to specify the semantics of object-oriented systems. [REN05]

Graph transformation rules specify changes of a small sub-structure. These changes represent modifications to these small sub-structures like deleting parts from it or adding parts to it. In order for the rule to apply, the host graph (graph that is subject to transformation) is required to contain a sub-structure that matches the graph transformation rule.

A graph consists of a set of nodes and a set of edges with labels which represent names of the edges. These edges connect two nodes to each other.

Figure 2.6 shows an example of a graph. This example represents a circular buffer with two pointers, which point to the first and the last cell of the buffer. The circular buffer contains one value.
Figure 2.6 - graph of circular buffer

Figure 2.7 shows a graph transformation rule of a get-operation on a circular buffer. Thin blue dashed edges and nodes mean that these components have to be deleted and green thick edges and nodes mean that these components will have to be added to the graph. So with this get-operation, the first value in the buffer is deleted, and the pointer is moved to the next cell.

Figure 2.7 – example graph transformation rule – get operation on circular buffer

A graph transformation rule consists of a left-hand-side (LHS) and a right-hand side (RHS). In Figure 2.7 everything except the thick green line represents the LHS. The RHS is represented by everything except the blue dashed edges and nodes.

A graph transformation rule is applicable to a graph A if there is a matching of LHS into graph A.

Figure 2.8 shows the matching of the get-operation with thick lines. The buffer node with the edge first towards the cell node is matched. The val edge towards the object node is matched. And the next edge towards the cell node is matched.
When applying a rule, the nodes and edges that are in the LHS, but not in the RHS will be deleted from graph A (these are the thin blue dashed edges and nodes in the figures in this thesis) and the nodes and edges that are in the RHS, but not in the LHS will be added to graph A (these are the thick green edges and nodes in the figures in this thesis).

Figure 2.9 shows the resulting graph after applying the get-operation to the graph of Figure 2.8. It can be seen that the \textit{val} edge towards the \textit{object} node is deleted. Furthermore the \textit{first} edge is moved to the \textit{next} cell node.

Furthermore there can be negative application conditions, which can say for instance that a certain node cannot have a certain outgoing edge. If the node does have an edge, the rule does not apply there. Figure 2.10 shows the negative application condition as thick dashed red edges and nodes. This figure shows the put-operation of a circular buffer. The last-pointer is moved one cell further, and this cell will point to the object being put. This can only happen if this cell does NOT have a pointer to another object already. In other words, the buffer cannot be full already.
2.5 Graph transformations for model transformations

OMG requested that a QVT language must have some characteristics:

- horizontal (PSM to PSM, see section 2.1) and vertical (PIM to PSM, see section 2.1) model-transformations should be supported
- model transformation rules should be automatically applied.
- transformation rules have to be easy to understand
- transformation rules have to be adaptable and reusable

[OMG02b, GRU05]

Grunske et al. [GRU05] propose to use graph transformations for model transformations, based on these requirements. They state the following reasons for this:

- Graphs are a natural representation for models, because modelling languages are usually formalized by a visual abstract syntax definition.
- An established formal theory and formalisms for automatic application are provided.
- Graph transformation rules are easy to specify. Although there are no empirical studies to prove this.
- The complexity of graph transformation rules and the application formalisms can be hidden for the end user.

Küster et al. [KUS04] compare a graph transformation based approach and a relational approach based on a QVT submission. Comparisons are made under different criteria:

- One comparison is made from a software-engineering point of view. Here they compare the ease and effort for designing model transformations. They observe that both approaches deal with approximately the same amount of rules. Also the models in the rules are quite similar. They see a slight difficulty for compound rules for graph transformations. Furthermore they state that usability depends on the user’s background and requires extensive empirical studies before making conclusions.
- Another comparison is made from the viewpoint of computation. They show that both approaches have advantages and disadvantages, but do not favour one.
Lastly it is stated that the graph transformations approach do not support bi-directionality usually, while the relational approach does.

An example of a program which uses graph transformations to perform model transformation is FUJABA. FUJABA is a program which translates UML into Java and Java into UML (the Java into UML part is still in development though), using graph transformations. It uses a combination of activity diagrams and collaboration diagrams, called story diagrams, to represent graph transformations. Nodes are represented as objects and edges as associations. Objects and links can be marked with stereotypes like <<destroy>> and <<create>> to delete or create nodes and edges. [NIE99]

The goal of this thesis is to use graph transformations, in order to be able to transform between any arbitrary meta-models. FUJABA only supports the transformation between UML and Java and is not capable to perform transformations for arbitrary meta-models.

### 2.6 Conclusion

Section 2.1 showed what meta-models are and what MOF is. After this, section 2.2 gave an overview of model transformation in MDA and what QVT is. Section 2.3 gave an introduction on ITL and showed the grammar of this proposal. Then section 2.4 described what graph transformations are and how they work, followed by section 2.5 which showed how graph transformations are used for model transformation.
3 Classification of ITL and graph transformation rules

After having seen a description of ITL in section 2.3 and a description of graph transformations in section 2.4, this chapter will try to compare the two approaches. This way it can be seen if graph transformations can be used instead of ITL. Czarnecki [CZA03] has analyzed the variability’s and commonalities of existing model transformation approaches. Using feature diagrams he represents areas of variation. This chapter classifies ITL (the IBM Transformation Language) [DSTC04] according to the classification of Czarnecki in order to see what the characteristics are of this proposal. This is followed by seeing how graph transformations can represent the same characteristics as ITL. The presented classification figures in this chapter have been taken from Czarnecki [CZA03].

3.1 Description classification and ITL classified

This section gives a description about the classification system in Czarnecki [CZA03] and how ITL fits into this classification.

Figure 3.1 shows the major areas of variation for model transformation. Mandatory features are represented by a black dot and optional features are represented by a white dot, as can be seen in the legend in Figure 3.2.

---

Figure 3.1 – feature diagram of major areas of variation

Legend:

- ○ mandatory
- □ optional
- ← alternative
- ▽ inclusive or

Figure 3.2 – legend of feature diagrams
All major areas of variation are discussed shortly hereafter. It is also discussed how ITL fits into this.

### 3.1.1 Transformation rules

Transformation rules have left-hand sides (LHSs) and right-hand sides (RHSs). The LHS represents something in the source model and the RHS represents something in the target model. Figure 3.3 shows the classification possibilities for transformation rules. The possibilities will be discussed now.

**Syntactic separation**

Syntactic separation means that the rule syntax separates the RHS from the LHS. In ITL the LHS and RHS are syntactically separated, because there is a clear distinction between the two. As can be seen in Appendix B, the LHS in a `tRule` leads to other nonterminals than the RHS. The RHS can perform assignments, which is not allowed in the LHS.

**Bidirectionality**

Bidirectionality means that a rule may be executable in the inverse direction.
ITL only supports transformation from the LHS to the RHS model. Bi-directionality is not supported directly. To transform back a separate transformation definition needs to be created.

**Parameterization**
Transformation rules of ITL may have parameters being passed when a rule is being called. This could allow giving control parameters for configuration and tuning.

**Intermediate structures**
Some approaches may require intermediate structures being created. ITL does not require any intermediate structures though.

**Variables**
Variables can hold elements of the source and target models. When a variable is syntactically typed, this means that a variable is associated with a meta-model element whose instances it can hold. Semantic typing can require more properties. A syntactic type example could be `Expression`, and the semantic type could be `Expression evaluating to an integer value`.

Transformations in ITL can hold local `tRule` variables and global variables within the whole transformation. The variables in ITL are syntactically typed.

**Logic**
Logic expresses computations and constraints on model elements. None-executable logic only specifies relationships between models. A declarative language is the opposite of an imperative language. An imperative language specifies explicit sequences of steps to follow to produce a result. A declarative language describes relationships between variables in terms of functions and the compiler applies a fixed algorithm to these relations to produce the target.

The logic in ITL is declarative and executable. This is because of the implicit creation of target elements through constraints.

**3.1.2 Rule application scoping**

![Rule Application Scoping](image)

Figure 3.4 - features of rule application scoping
Source scope means that the scope of the source model is considered for rule application. Target scope means that the target model is considered as scope, where the RHS will be expanded.

ITL has source scope as rule application scoping, since the scope of the source model is taken in order to apply rules.

### 3.1.3 Relationship between source and target

ITL requires the creation of a new target model, separated from the source. This is in contrary to approaches that update the source model or make adjustments to the source in order to reach its goal.

### 3.1.4 Rule application strategy

Rules need to be applied to specific locations within their scope. Because there may be more than one match for a rule, an application strategy is required. A deterministic strategy exploits some standard traversal strategy. A non-deterministic strategy can apply rules on locations chosen in a non-deterministic chosen order (one-point) or it can apply...
rules concurrently to all matching locations (concurrent). Furthermore the target can also be chosen interactively by a user.

ITL supports traceability links. These can be used in other transformation rules to determine the target, by following the traceability link of a source element that was created by some other rule. Furthermore ITL is one-point non-deterministic.

3.1.5 Rule scheduling
Rule scheduling determines the order in which rules are executed. Figure 3.7 shows the classification possibilities for rule scheduling. The possibilities will be discussed now.

Form
ITL supports internal explicit scheduling, because it allows transformation rules to directly invoke other rules.

Rule selection
The order in which rules are applied can be predefined or it can be non-deterministic. Rule scheduling in ITL is non-deterministic. [DSTC04]
3.1.6 Rule organization

Figure 3.8 shows the classification possibilities for rule organization. The possibilities will be discussed now.

Modularity mechanisms
In ITL, transformation rules (tRules) are packaged together in a transformation. This is the only form of modularity mechanism presented in ITL.

Reuse mechanisms
ITL implements reuse mechanisms. A transformation rule can extend or supersede other transformation rules. Extending a rule means that the rule only applies to elements that match both the extender and the extended rule. Superseding a rule means that the superseded rule applies only to elements the superseder rule do not apply to.

Organizational structure
Rules can be source-oriented or target-oriented. These structures have one rule for each element and are nested according to the hierarchy in the meta-model.

Rules in ITL are not organized according to the structure of the source or target language. Therefore the organizational structure could be called independent.
3.1.7 Traceability links

A transformation may create links between its source and target elements. This is needed for determining the target of a transformation for instance.

ITL provides dedicated support for traceability. Creation of traceability links need to be encoded manually in the transformation rules.

3.1.8 Directionality

When transformations can only transform from source to target, these transformations are called unidirectional. Transformations that can transform in both directions are called bi-directional.

ITL only supports transformation from the LHS to the RHS model. Bi-directionality is not supported directly. To transform back a separate transformation definition needs to be created.

3.1.9 Conclusion

ITL falls in the major category “Relational approaches” [CZA03]. This category consists of declarative approaches with as main concept mathematical relations. According to Czarnecki “relational approaches seem to strike a well balance between flexibility and declarative expression. They provide flexible scheduling and good control of nondeterminism.” [CZA]

3.2 Graph transformations classified

This section discusses the classification of Czarnecki [CZA03] to compare ITL to the world of graph transformations.
3.2.1 Transformation rules

**Syntactic separation**
Unlike ITL, there is no clear syntactic separation between the LHS and the RHS in graph transformations in general. In some graph notations which combine the LHS and the RHS in one graph, a distinction could be made, because they have different notations for deletions (RHS) and negative application conditions (LHS).

**Bidirectional**
Graph transformations in general support both the transformation from the LHS to the RHS model as well as bidirectional transformations. In some graph notations which combine the LHS and the RHS in one graph, only transformation from the LHS to the RHS model is supported.

**Parameterization**
Parameters might be passed to transformation rules by having a node ‘Parameters’ with edges that represent the parameters.

**Intermediate structures**
Like ITL, graph production systems do not require intermediate structures either.

**Variables**
Graph transformations in general do not support variables. Global variables would require a special node `GlobalVariables` to be made with edges that represent the global variables.

**Logic**
Graph transformations could be classified as a declarative language, just like ITL. Relationships are described between the nodes and edges and the compiler applies a fixed algorithm to these relations to produce the target.

3.2.2 Rule application scoping
Graph transformation rules provide source scoping, like ITL. Nodes and edges in the source model are considered in order to apply rules.

3.2.3 Relationship between source and target
In contrast to the approach by ITL, graph transformation rules generally update the source model when rules are applied instead of creating a new target model.
3.2.4 Rule application strategy
Traceability links are not directly supported by graph transformation rules. This could be emulated by creating a node Link with a source-edge and a target-edge.

3.2.5 Rule scheduling

Form
Graph transformations support system-defined implicit scheduling. In order to emulate internal explicit scheduling as in ITL, the same kind of emulating as in Rule application strategy could be used. This way a rule could be made to be executed only after another rule created a node Link.

Rule selection
Rule scheduling for graph transformations is non-deterministic.

3.2.6 Rule organization

Modularity mechanisms
In graph production systems there are no modularity mechanisms. The only module available is a graph transformation rule. As compared to ITL, a TRule in ITL could be compared to a graph transformation rule. A transformation in ITL can be compared to a graph production system.

Reuse mechanisms
Graph transformation rules cannot be superseded or extended.

Organizational structure
Just like ITL, the organization structure for a graph production system could be called independent. Graph transformation rules are not organized at all.

3.2.7 Traceability links
Creating traceability links for graph transformations can be done by copying the source model in the target model and creating a node Link that has edges to this copied source-model and the created target-model.

3.2.8 Directionality
Graph transformation rules supports directionality from LHS to RHS as well as bi-directionality.
3.3 Conclusion

Section 3.1 showed how ITL can be classified according to Czarnecki’s classification. Section 3.2 then showed how graph transformations compare to ITL, based on this classification. This comparison has been summarized in Table 1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ITL</th>
<th>Graph transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation rules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Syntactic separation</td>
<td>V</td>
<td>X</td>
</tr>
<tr>
<td>• Bidirectionality</td>
<td>X</td>
<td>V</td>
</tr>
<tr>
<td>• Parameterization</td>
<td>V</td>
<td>Possible</td>
</tr>
<tr>
<td>• Intermediate structures</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>• Variables</td>
<td>V syntactically typed</td>
<td>Possible</td>
</tr>
<tr>
<td>• Logic</td>
<td>Declarative executable</td>
<td>Declarative executable</td>
</tr>
<tr>
<td>Rule application scoping</td>
<td>Source</td>
<td>Source</td>
</tr>
<tr>
<td>Relationship between source and target</td>
<td>New target</td>
<td>Update existing model</td>
</tr>
<tr>
<td>Rule application strategy</td>
<td>One-point non-deterministic</td>
<td>One-point non-deterministic</td>
</tr>
<tr>
<td>Rule scheduling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Form</td>
<td>Internal explicit</td>
<td>System-defined implicit</td>
</tr>
<tr>
<td>• Rule selection</td>
<td>Non-deterministic</td>
<td>Non-deterministic</td>
</tr>
<tr>
<td>Rule organization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Modularity mechanisms</td>
<td>Transformation rules</td>
<td>Graph transformation rules</td>
</tr>
<tr>
<td>• Reuse mechanisms</td>
<td>Inheritance</td>
<td>X</td>
</tr>
<tr>
<td>• Organizational structure</td>
<td>Independent</td>
<td>Independent</td>
</tr>
<tr>
<td>Traceability links</td>
<td>Manual</td>
<td>Manually possible</td>
</tr>
<tr>
<td>Directionality</td>
<td>Unidirectional</td>
<td>Unidirectional and bidirectional</td>
</tr>
</tbody>
</table>

Table 1 – classification comparison of ITL and graph transformations

The main things that graph transformations cannot do, which ITL can do are:

• The source model is updated instead of a new target model being created.
• Inheritance of a graph transformation rule is not supported.
Furthermore some of the features are not directly supported, but can be emulated.

• Parameters might be passed to transformation rules by having a node ‘Parameters’ with edges that represent the parameters.
• Global variables would require a special node `GlobalVariables` to be made with edges that represent the global variables.
• Creating traceability links for graph transformations can be done by copying the source model in the target model and creating a node `link` that has edges to this copied source-model and the created target-model.
4 Problem analysis

This chapter describes what has to be done in order to solve the problem of this thesis.

In Figure 4.1 a subset can be seen of the structure of MDA as used by the QVT proposals presented in Figure 2.5.

This same kind of structure can be used in a graph transformation process. This structure is shown in Figure 4.2.

The corresponding parts of the structures represent the same idea. A meta-model for the MDA structure should represent the same as a type graph for a graph production system, etc.

A type graph defines the edges and nodes, which an instance graph can contain. An instance graph can only contain nodes, which are also in its type graph. It can also only contain edges between nodes, when these edges between these nodes are also present in its type graph.
In the QVT proposal a meta-model A and B, a transformation definition between these two and a model A are given, and a model B is the output.

The goal of this thesis is to have these same inputs and get the same output, but let graph transformation rules do the transformations. This is depicted in Figure 4.3, which represent the conversions between Figure 4.1 and Figure 4.2. The meta-models have to be converted to type graphs (1 in Figure 4.3), the source model A has to be converted to a source instance graph A (2 in Figure 4.3), and the transformation definition has to be converted to graph transformation rules (4 in Figure 4.3). Then using the source instance graph A, graph transformation rules and type graphs, the target instance graph B can be generated by applying the graph transformations. Finally the target instance graph B is converted to the target model B (7 in Figure 4.3) to get the same result as the MDA process.

![Figure 4.3 – conversion from MDA structure to graph production system structure](image)

In Figure 4.4 a data flow diagram can be seen that represents the order of actions that have to be taken in order to reach this goal.

As can be seen there are several steps needed to get the output model B. The following steps can be seen (the numbers correspond to the numbers in the data flow diagram):

1. The given meta-models have to be converted somehow to type graphs. These meta-models should be provided as a MOF file.
2. The given source model has to be converted to an instance graph. This model should also be provided as a MOF file.
3. The given A to B transformation definition has to be scanned, parsed and analyzed in order to create a syntax tree.
4. The syntax tree needs to be used in order to make a translation to graph transformation rules from an arbitrary model A to a model B. The target meta-models B is also needed in this step to provide bidirectionality.
5. Using the graph transformation rules and the instance graph of the source model, a graph transformation can be applied which gives a combined instance graph with backtracking information of the source and target model.
6. The instance graph of the target model has to be extracted from the combined instance graph.
7. This target instance graph needs to be converted to a model of the target eventually. The target meta-model will be needed for this conversion to provide bidirectionality.

The grey oval areas in Figure 4.4 represent the areas for which a general conversion-process has to be constructed in this thesis.

First the grammar of ITL (the IBM Transformation Language) has to be studied. Encountered problems will be shown in chapter 5. After this an example transformation has to be manually converted to graph transformations. This will be presented in chapter 6. After this a general way has to be devised in order to create graph transformations for any arbitrary transformation definition. This will be explained in chapter 7. Finally a tool has to be built in order to proof this general way. Chapter 8 presents the tool.
Figure 4.4 – dataflow diagram of conversion from QVT to graphs
5 Problems encountered in ITL

ITL (the IBM Transformation Language) has a language definition, which has been described in section 2.3. ITL also presents an example transformation (a copy can be found in Appendix A of this thesis). In order to apply the solutions presented in chapter 4, the grammar has to be understood and the examples have to be parsed according to this grammar. In order to understand how the ITL language works, the example transformation has been tried to be parsed according to the presented grammar. So the examples given in ITL have been studied and a concrete syntax tree has been tried to create. From this concrete syntax tree, an instance of the transformation model was tried to create.

Doing this it turned out that the examples, concrete syntax and transformation model were not consistent and also not described clearly. It seems that this is not only the case for ITL, but also for the other proposals. According to Gardner et. al [GAR03] many submissions were found to be incomplete and unclearly formulated. Also they assume it is hard to adopt any of the proposals. They also mention that ITL can only be fully exploited after having understood the hundred pages long paper. Furthermore they mention that ITL is not really self-contained and that it refers to other languages described elsewhere. The given examples are either highly simplified or nonexistent and therefore only a limited picture can be obtained of how a transformation would be represented and executed [GAR03]

5.1 Problems in the grammar

Trying to make a concrete syntax tree of the examples of ITL, using the EBNF Concrete syntax grammar of ITL (a copy can be found in Appendix B of this thesis), it became clear that the grammar was not correct nor complete. In this chapter some findings are shown that show the incorrectness of the given grammar. The findings are categorized per nonterminal of the grammar. Also solutions are proposed and a modified grammar can be found in Appendix B.

5.1.1 tRule

The grammar of nonterminal tRule is as follows:

```
tRule : "RULE" rname formals
( relatedRules )*
"FORALL" ranges
( "WHERE" conjunct )?
"MAKE" targets
( "LINKING" trackingUses )?
SEMI
```

Problem 1

Where
In the example that can be found in Figure 3.5 of Appendix the following part can be found:
WHERE JavaClassFromUMLClassifier jcuc AND ....

What
It is impossible to transit from nonterminal conjunct to tname vname AND ....

Possible solution
A possible solution to overcome this problem would be to add \( \text{tname vname} \) to nonterminal relation.
It is unclear though, why this reference to a trackingUse is in the WHERE-part, since it only binds the variable \( \text{jcuc} \). Furthermore it is unclear what something like WHERE (\( \text{NOT JavaClassFromUMLClassifier jcuc} \)) AND ... would mean.
Therefore it is better to put this reference in the \text{FORALL}-part of a tRule. To avoid nondeterminism also a keyword is needed to place in front of the reference to the trackingUse.
So the most logical solution would be to add \( \text{\textasciitilde TUSE tname vname to nonterminal range} \).

Problem 2

Where
In the example that can be found in Figure 3.5 of Appendix the following part can be found:
AND JavaClass jc = jcuc.javaClass

What
It is impossible to transit from nonterminal conjunct to FACTOR ASSIGN FACTOR, since JavaClass jc cannot be a FACTOR

Possible solution
A possible solution to overcome this problem is to add \( \text{range to factor} \)

Problem 3

Where
In the example that can be found in Figure 3.7 of Appendix the following part can be found:
FORALL UMLClassifierAndName(uc, n)

What
It is impossible to transit from nonterminal ranges to patternUse

Possible solution
A possible solution to overcome this problem is to add \( \text{patternUse to range} \)
Problem 4

Where
In the example that can be found in Figure 3.7 of Appendix the following part can be found:
MAKE JavaClassAndName(jc, n)

What
It is impossible to transit from nonterminal target to patternUse

Possible solution
A possible solution to overcome this problem is to add | patternUse to target

Problem 5

Where
In the example that can be found in Figure 3.9 of Appendix the following part can be found:
MAKE jp1 BEFORE jp2 IN m.arg

What
It is impossible to transit from nonterminal target to linkOrder

Possible solution
A possible solution to overcome this problem is to add | linkOrder to target

Problem 6

Where
In the example that can be found in Figure 3.9 of Appendix the following part can be found:
RULE umlParameterToJavaParameter FORALL ........

What
LBRACK and RBRACK are required after umlParameterToJavaParameter

Possible solution
A possible solution to overcome this problem is to change the example to
RULE umlParameterToJavaParameter() FORALL ........
Problem 7

Where
In the grammar of ITL.

What
tRule contains the part to ( “LINKING” trackingUses)?
trackingUse leads to “LINKING” tname vname “WITH” assignment (COMMA assignment) *

This leads to twice the use of the word “LINKING”.

Possible solution
scrap the “LINKING” word from tRule.

Problem 8

Where
In the grammar of ITL.

What
nonterminals relation and target can lead to a link.
nowhere is defined what a link is.

Possible solution
Since it is not clear what the link has to be, the link is scrapped for this thesis.

5.1.2 patternDefn
The grammar of nonterminal patternDefn is as follows:

```
patternDefn
  : "PATTERN" pname formals
  ( "FORALL" ranges
  ( "WHERE" conjunct )?
  | "MAKE" targets
  )
  SEMI
```

Problem 1

Where
In the example that can be found in Figure 3.7 of Appendix the following part can be found:
MAKE javaclass jc AND jc.name = name

What
It is impossible to transit from nonterminal target to range AND assignment

**Possible solution**
A possible solution to overcome this problem is to change the example to
MAKE javaclass jc , jc.name = name

Problem 2

**Where**
In the example that can be found in Figure 3.7 of Appendix the following part can be found:
DEFINE PATTERN.....

**What**
The word DEFINE does not appear in nonterminal patternDefn

**Possible solution**
A possible solution to overcome this problem is to add the word DEFINE in front of nonterminal patternDefn

5.1.3 linkOrder
The grammar of nonterminal linkOrder is as follows:

```plaintext
linkOrder :vname "BEFORE" vname "IN" vname 
```

**Problem 1**

**Where**
In the example that can be found in Figure 3.9 of Appendix the following part can be found:
MAKE jp1 BEFORE jp2 IN m.arg

**What**
It is impossible to transit from nonterminal vname to m.arg, which is a path

**Possible solution**
A possible solution to overcome this problem is to change vname to (vname | path) in LinkOrder
5.2 Problems in the abstract syntax meta-model

ITL shows an abstract syntax meta-model in their proposal and there should be a connection to the concrete syntax grammar. The abstract syntax meta-model can be seen in Figure 5.1 [DSTC04].

Figure 5.1 – ITL abstract syntax meta-model
Trying to create an instance of the concrete syntax tree of the examples, using the abstract syntax meta-model of ITL, it became clear that the link between the grammar and the abstract syntax meta-model was incomplete and not clear. In this chapter some findings are shown that show the incorrectness and vagueness of the given grammar. Suggested solutions are given, but the abstract syntax meta-model is not used for the remainder of this thesis due to its vagueness and because it is not required to continue the work of this thesis.

5.2.1 Assignment

Where
Nonterminal assignment leads to path ASSIGN factor.

Possible solution
The only object that seems appropriate is a Condition with as argument an Expression which represents path = factor. This is represented in Figure 5.2.

Unresolved issues
An assignment is not really a condition though, but it can be seen as “this path must be equal to this factor, in order to let this tRule succeed”.
Also it is questionable how to represent the Expression. A StringConstant with path = factor takes everything into one string, which does not really seem perfect.

5.2.2 trackingUse

Where
Nonterminal trackingUse leads to LINKING tname vname WITH assignment (COMMA assignment)*

Possible solution
It seems straightforward to create an object TrackingUse. This object has a relation tracking to a MOF:Class, with tname as name.
For every assignment there should be an ordered relation of featurenames and a corresponding ordered relation of arguments. The featurenames must be StringConstants, which represent the path of the assignments. The arguments must be Expression that represents the factor of the assignments
Further a TrackingUse object must have one target to a VarUse which points to ‘the target model element that functionally depends on the source model elements’ [DSTC04]. This possible solution has been depicted in Figure 5.3.

Unresolved issues
It is not really clear what the target VarUse should point to. Also unclear is how to represent the vname in the object-model.

5.2.3 patternDefn

Where
Nonterminal patternDefn leads to PATTERN pname formals (FORALL ranges (WHERE conjunct)?) | MAKE targets) SEMI

Possible solution
It seems straightforward to create an object PatternDefn with pname as name. This object has formals as parameter-relations.
A PatternDefn has exactly one body which must be a Term.
It is unclear how to represent the FORALL ranges and WHERE CONJUNCT or MAKE targets in this one Term.
The closest representation would be an AndTerm which can contain other Terms to represent the different aspects. It does not seem to be possible to make distinction between the different parts though.

Unresolved issues
It is unclear how to represent the FORALL ranges and WHERE CONJUNCT or MAKE targets in one AndTerm.
5.2.4 patternUse

Where
Nonterminal patternUse leads to pname actuals
In the example of ITL, which can be seen in Figure 3.7 of Appendix of this thesis the following patternUse can be seen:
UMLClassifierAndName(uc,n )

Possible solution
A Patternuse object is created which has the actuals as argument and has a pointer to the PatternDefn with pname as name.

Unresolved issues
n is a new variable that is introduced in this line for the first time. So it is unclear what this VarUse should point to.

5.2.5 Path

Where
Nonterminal path leads to vname (PERIOD feature)+

Possible solution
The most appropriate way seems to be to create an object FeatureExpre with as attribute a featureName, and with a relation argument to the object of vname.

Unresolved issues
It is unclear how to represent a path.

5.2.6 WHERE part in tRule

Where
Nonterminal tRule can have a WHERE conjunct part to restrain the source defined by FORALL ranges.

Possible solution
What seems most appropriate is to have two source-relations from a TRule object. One that relates to the ‘FORALL ranges’-part, and one to the ‘WHERE conjunct’-part.

Unresolved issues
It is unclear how to represent the ‘WHERE conjunct’ and ‘FORALL ranges’ parts in a TRule object
5.2.7 Range

Where
Nonterminal range leads to mofType vname, which leads to ID ID.

Possible solution
It seems most appropriate to create a MOFInstance with as instance a VarUSE which points to the AbstractVar with vname as name.
The typename of a MOFInstance should be an Expression. The only kind of expression that seems appropriate is a StringConstant with as representation the name of the mofType.

Unresolved issues
These are just guesses though. A StringConstant for the mofType does not seem really perfect.

5.3 Conclusion
The encountered problems in the translation from the example to a syntax tree, mentioned in section 5.1, could be fixed either by changing the syntax grammar or by changing the examples in ITL. By applying the given proposed possible solutions in this thesis, the encountered problems should be fixed. The only question that remains then is whether or not the meaning of the authors of ITL is still valid.

The problems with translating syntax trees to instances of the transformation model of ITL, mentioned in section 5.2, are trickier. The translation is not at all straightforward and the problems mentioned in section 5.2 are not as easy to fix as the problems mentioned in section 5.1. The possible solutions given are based on guesses, of which some still have unclearness left.

Therefore it is probably best not to depend on the transformation model too much for this thesis’s work. Since it does not represent something else than the grammar, but just shows the modular structure of the grammar, the work in this thesis does not have to suffer much from the incomplete understanding of this model.

In Appendix B a modified concrete syntax grammar can be found that fixes problems mentioned in section 5.1 and should be a complete working grammar. This modified grammar is used in the remainder of this thesis.
6 Example transformation

There is an example transformation in ITL (the IBM Transformation Language), which transforms a UML model to a Java model. It is used to try out the basic approach of conversion to graph transformations. After applying the solutions presented in chapter 5, the example transformation definition of ITL is taken and this chapter gives the representation as graph transformations according to the scheme presented in Figure 4.4. All grey ovals of Figure 4.4 will be discussed in this chapter.

Section 6.1 shows how the source and target meta-models are converted to type graphs (11). After this section 6.2 shows the conversion of the transformation definition of ITL to graph transformation rules (3 & 4). Then section 6.3 will show the conversion of an example source model to an instance graph (2). Section 6.4 will then discuss the application of the graph transformation (5). Section 6.5 will extract the target instance graph out of the combined instance graph that is created (6). Finally section 6.6 will convert the target instance graph to the target model (7).

6.1 Meta-model to type graph

This paragraph converts the meta-models of UML and JAVA to type graphs.

6.1.1 UML

Figure 6.1 shows a simplified UML meta-model, shown in ITL.

![Figure 6.1 – simplified UML meta-model](image)

The meta-model of UML needs to be converted to a type graph of UML. The resulting type graph can be seen in Figure 6.2.

---

1 Numbers between brackets refer to the numbers used in the diagram of Figure 4.4.
Classes have been converted to nodes, attribute types have been converted to nodes, and relations and attribute names have been converted to edges. Inheritance is left intact as inheritance edges in the type graph. Bidirectional relations have been converted into two edges, of which one edge for each role name.

### 6.1.2 Java

Figure 6.3 shows a simplified Java meta-model, taken from ITL.
The meta-model of Java must also be converted to a type graph of Java. The resulting type graph can be seen in Figure 6.4.

Also here classes have been converted to nodes, attributetypes have been converted to nodes and relations and attributenames have been converted to edges. There are two role names called owner between JavaMethod and JavaClass. Therefore both edges have been renamed with the name of the inverse edge as suffix. Multiple nodes String can be seen in the type graph. These represent the same node, but multiple nodes are represented to prevent too many crossing lines in the visual type graph.

6.2 ITL Rules to graph transformation rules

In the following sections, the rules of the example of ITL have been converted to graph transformation rules. The original code of the examples of ITL with corresponding names can be found in Appendix. The presented example code in this section has been modified where necessary due to incompleteness or incorrectness of the examples in ITL. The resulting modified example allows to present a working example later.
6.2.1 UMLClassifierToJavaClass

A rule to transform a UMLClassifier to a Java class is represented as follows in ITL language:

RULE UmlClassifierToJavaClass(uc, jc)
FORALL UMLClassifier uc
MAKE JavaClass jc,
jc.name = uc.name
LINKING JavaClassFromUMLClassifier jcuc
WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;

This rule checks for a UMLClassifier and creates a JavaClass, and assigns it the name of the UMLClassifier. Furthermore, a trackingUse is created which links the source UMLClassifier with the target JavaClass.

This rule can be converted to the graph transformation rule presented in Figure 6.5.

![Figure 6.5 – graphrule UmlClassifierToJavaClass](image)

The graph transformation rule checks for a node UMLClassifier with a name and creates a JavaClass node with an edge to this same name. A LINK is created to provide traceability between the source and target and to bind variable names to the nodes. Also the link is used to prevent rules from being executed...
twice. This is done by using the negative application condition that an UMLClassifier cannot have a LINK node of the current rule name already incoming.

6.2.2 UMLAttributeToJavaField

A rule to transform a UMLAttribute to a JavaField is represented as follows in ITL language (The creation of the name of the UMLAttribute is not mentioned in the original example in ITL):

RULE UmlAttributeToJavaField(ua,jf)
FORALL UMLAttribute ua, TUSE JavaClassFromUMLClassifier jcuc
WHERE jcuc.umlClassifier = ua.owner AND
JavaClass jc = jcuc.javaClass
MAKE JavaField jf,
jf.owner = jc,
jf.name = ua.name
LINKING FieldFromAttr ffa
WITH ffa.javaField = jf, ffa.umlAttr = ua;

The rule checks for a UMLAttribute and a trackingUse, created by the rule mentioned in section 6.2.1, of which the owner of the UMLAttribute equals the UMLClassifier of the trackingUse. It then stores the corresponding JavaClass in variable jc. After this a JavaField is created with as owner the JavaClass jc and as name the name of the UMLAttribute. Furthermore a trackingUse is created again.

This rule can be converted to the graph transformation rule presented in Figure 6.6.

The graph transformation rule checks for a node UMLAttribute with a certain name and a certain UMLClassifier as its owner. This UMLClassifier must be part of a trackingUse node already, which should have been created by the rule presented in 6.2.1 already. A new JavaField is created which has the same name as the UMLAttribute. An owner edge is created to the JavaClass that is linked by trackingUse JavaClassFromUMLClassifier to the UMLClassifier. Also a field edge is created from the JavaClass to the JavaField. This edge is not mentioned in the code of the example in ITL, so has to be extracted from the meta-model. Also in this example a LINK is created for traceability and to prevent the rule from being executed more than once on a node.
6.2.3 UMLOperationToJavaMethod

A rule to transform a UMLOperation to a JavaMethod is not represented in ITL, but could be written as follow:

RULE UmlOperationToJavaMethod(uo, jm)
FORALL UMLOperation uo, TUSE JavaClassFromUMLClassifier jcuc
WHERE uo.owner = jcuc.umlClassifier
MAKE JavaMethod jm,
    jm.name = uo.name,
    jm.methodOwner = jcuc.javaClass
LINKING JavaMethodFromUMLOperation jmuo
WITH jmuo.javaMethod = jm, jmuo.umlOperation = uo;

This rule checks for a UMLOperation and a trackingUse, as created in section 6.2.1. The owner of the UMLOperation should equal the UMLClassifier of the trackingUse. After this a JavaMethod is created with as name the name of the UMLOperation, and as methodOwner the JavaClass of the looked up trackingUse. Finally a trackingUse is created again for this rule.
This rule can be converted to the graph transformation rule presented in Figure 6.7.

The graph transformation rule checks for a node UMLOperation with a certain name and creates a JavaMethod node with an edge to this same name. The trackingUse JavaClassFromUMLClassifier is used to identify the owner javaClass of the new javaMethod. Furthermore the LINK node and trackingUse node are used the same as in the previous graph transformation rules.

### 6.2.4 UMLClassToJavaClass

A rule to transform a UMLClass to a JavaClass is represented as follow in ITL language:

```itl
RULE UmlClassToJavaClass(uc, jc)
EXTENDS UmlClassifierToJavaClass(uc, jc)
FORALL UMLClass uc
MAKE JavaMethod m,
   m.name = uc.name,
   jc.constructor = m
LINKING JavaConsFromUMLClass consFromClass
```

Figure 6.7 – graphrule UmlOperationToJavaMethod
WITH
consFromClass.constructor = m,
consFromClass.umlClass = uc;

This rule refines the rule UmlClassifierToJavaClass, presented in section 6.2.1. It checks for a UMLClass this time, instead of the more general UMLClassifier. In addition to the rule UmlClassifierToJavaClass, it now also creates a JavaMethod with as name the name of the UMLClass. This JavaMethod represents the constructor, which is not present in the UML model.

This rule can be converted to the graph transformation rule presented in Figure 6.8.

The graph transformation rule checks for a node UMLClass with a certain name and creates a JavaClass node with an edge to this same name. Also a JavaMethod is created with this same name. The created JavaClass gets a constructor edge to this JavaMethod. Also an edge constructorOwner has to be created in the inverse direction although this is not explicitly mentioned in the code example.
Furthermore the LINK nodes and trackingUse nodes are created again.
All the work that the rule `UMLClassifierToJavaClass` does has been implemented also in this rule, because this rule extends the rule `UMLClassifierToJavaClass`. The rule `UMLClassifierToJavaClass` is therefore not useful anymore with this rule for UMLClasses.

### 6.2.5 UMLParameterToJavaParameter

A rule to transform a `UMLParameter` to a `JavaParameter` is represented as follow in ITL language:

```itl
RULE umlParameterToJavaParameter()
  FORALL UMLParameter up, TUSE JavaMethodFromUMLOperation jmuo
  WHERE jmuo.umlOperation = up.behaviouralFeature
  MAKE JavaParameter jp,
    jp.name = up.name,
    jp.owner = jmuo.javaMethod
  LINKING JavaParamFromUMLParam jpu
  WITH jpu.javaParam = jp, jpu.umlParam = up;
```

This rule checks for a `UMLParameter` and a `trackingUse`, as created in section 6.2.3. The `UMLOperation` of this `trackingUse` should equal the `behaviouralFeature` association of the `UMLParameter`. Then a `JavaParameter` is created with the same name as the `UMLParameter`. The `owner` of the `JavaParameter` is the `JavaMethod` of the looked up `trackingUse`. Finally a new `trackingUse` is created for this rule.

This rule can be converted to the graph transformation rule presented in Figure 6.9.

The graph transformation rule checks for a node `UMLParameter` with a certain name and creates a `JavaParameter` node with an edge to this same name. The `trackingUse` `JavaMethodFromUMLOperation` is used to identify the owner `javaMethod` of the new `javaParameter`. Furthermore the `LINK` node and `trackingUse` nodes are used the same as in the previous graph transformation rules.
6.3 Source model to source instance graph

In Figure 6.10 an example is given of a UML-class.

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name : String</td>
</tr>
<tr>
<td>- address : String</td>
</tr>
<tr>
<td>+ eat() : void</td>
</tr>
<tr>
<td>+ sleep(time : String) : void</td>
</tr>
</tbody>
</table>

Figure 6.10 – example UML class

Figure 6.11 shows this UML-class represented as an instance of the UML meta-model.
This instance of the meta-model has to be converted to an instance graph. The instance graph that follows out of the instance of the meta-model can be seen in Figure 6.12.

The objects have been converted to nodes with the name of the object’s class as label. Variables are presented as edges pointing towards nodes with the variable’s value as label.

Relations between objects are presented as two edges pointing in inverse directions with the role names as label.
6.4 Applying graph transformation

When the graph transformation rules are applied to the UML instance graph of section 6.3, a combined instance graph is generated. This combined instance graph can be seen in Figure 6.13. It is a very big graph and cannot be displayed clearly therefore.
Figure 6.13 – combined instance graph

Figure 6.14 shows part of this combined instance graph zoomed in. It shows a LINK node, created by the rule umlParameterToJavaParameter, between a source UMLParameter and a target JavaParameter along with their trackingUse node JavaParamFromUMLParam. It also shows that this LINK node points towards the JavaMethodFromUMLOperation trackingUse node. This trackingUse was used to lookup the corresponding JavaMethod from the UMLOperation, which contains the UMLParameter. That way the edges between the JavaMethod and its corresponding JavaParameter could be made.
6.5 Extracting target instance graph out of combined instance graph

When all the `LINK` nodes and the `trackingUses` and the source model are deleted out of the combined instance graph of Figure 6.13, the model in Figure 6.15 is obtained.
6.6 Converting target instance graph to target model

The extracted target instance graph in Figure 6.15 has to be converted to a target model. The converted target model can be seen in Figure 6.16. The conversion is like the conversion from model to instance graph shown in section 6.3, but this the other way around.
6.7 Conclusion

This chapter showed an example transformation from UML to Java. This example was converted from ITL to graph transformations.

Section 6.1 showed how the source and target meta-models are converted to type graphs. After this section 6.2 showed the conversion of the transformation definition of ITL to graph transformation rules. Then section 6.3 showed the conversion of an example source model to a source instance graph. Section 6.4 then discussed the application of the graph transformation. Section 6.5 showed how to extract the target instance graph out of the combined instance graph that was created.

Finally section 6.6 converted the target instance graph to the target model.
7 General transformation

Chapter 6 showed all necessary steps for one example transformation from a UML model to a Java model. The goal of this thesis is to be able to deal with any arbitrary example though.

This chapter gives a general approach for all key parts of conversion from ITL (the IBM Transformation Language) to graph transformations. All grey ovals of Figure 4.4 are discussed in this chapter.

Section 7.1 treats the conversion from source and target meta-models to type graphs. (1\(^1\)) In section 7.2 the translation from the ‘A To B transformation definition’ of ITL to ‘Graph transformation rules A To B’ is discussed. (3 & 4)

Section 7.3 shows how to convert a model to an instance graph. (2)

Section 7.4 tells how to apply the graph transformation to get the combined instance graph. (5)

Section 7.5 shows how to extract the target instance graph from this combined instance graph (6)

Finally section 7.6 shows the conversion of an instance graph to the corresponding model. (7)

7.1 Converting meta-models to type graphs

It is assumed for this thesis that the meta-model only consists of classes with attributes that have a certain type and binary relations, without ordering constraints, between these classes. Inheritance is also possible between these classes. All MOF elements presented in section 2.1, including bidirectionality, are supported.

The following points show how to convert the elements in a meta-model to a type graph:

- Each class can be converted to a node with the name of the class as label.
- Each attribute can be converted to an edge with the attribute name as its label. This edge is pointed towards a node with the attribute type as its label. The source of the attribute edge is the node of the class that contains the attribute.
- A binary relation between two classes is converted to two edges, pointing in inverse directions and having the role name as label. When two role names on two different edges between two certain class-nodes happen to be the same, they should be renamed. This can be done by appending the inverse role name to the role name. If this still leaves identical role names, then a unique number could be appended.
- Inheritance between two classes is converted to an inheritance edge between the two created nodes of these classes.

Multiplicity has no impact on making graph transformations. It is only useful for verifying whether a certain model is correct in relation to its meta-model. Therefore it is not converted to the type graph.

\(^1\) Numbers between brackets refer to the numbers used in the diagram of Figure 4.4.
All these principles have been shown in the examples presented in section 6.1.

### 7.2 Converting ITL transformation rules to graph transformation rules

The concrete syntax grammar of a `tRule` is presented as follow in ITL [DSTC04]:

```plaintext
```

All subparts, indicated by the numbers in front of the lines, of this nonterminal are discussed now in order to create a general transformation system from a `tRule` to graph transformation rules.

#### 7.2.1 “RULE” rname formals

The name of a rule, `rname`, is needed to prevent rules from being executed more than once. Figure 7.1 shows a `LINK`-node being created with an edge `name` to a node with the `rname`.

![Figure 7.1 – graph transformation rule part rname formals](image)

Formals will be ignored here, since the variables introduced here in the transformation language of ITL, are repeated in the `FORALL` and `MAKE` part. So this information does not add anything to the information presented in the rest of the `tRule`.

#### 7.2.2 (relatedRules)*

nonterminal `relatedRules` leads to

```plaintext
( "extends"  
| "supersedes"
)
```

Rname formals (COMMA rname formals)*

**Extends**

The extended rule can be syntactically combined with the current rule in order to create a new `tRule` without related rules mentioned in it. This newly created rule in the format of ITL can then be converted to graph transformations.

The extended rule should not be applicable anymore where this rule applies.
**Supersedes**

The superseding rule can be converted normally to a graph transformation rule. The superseded rule should not be applicable anymore where this rule applies.

Figure 7.2 shows an example meta-model of a class A with subclasses B and C.

![Figure 7.2 – example meta-model](image)

The following code snippet will be converted to graph transformation rules:

```plaintext
Rule P
FORALL A
......
```

This code will be converted to three graph transformation rules. One rule for the class A (Figure 7.3), and two rules for its subclasses (Figure 7.4 and Figure 7.5).

![Figure 7.3 – graph transformation rule 1 for example 1](image)

![Figure 7.4 – graph transformation rule 2 for example 1](image)

![Figure 7.5 – graph transformation rule 3 for example 1](image)

The following code snippet which contains nonterminal `supersedes` will be converted to graph transformation rules also.
This code will be converted to the graph transformation rule presented in Figure 7.6.

The superseded rule should not be applicable anymore where rule Q applies. This goal can be reached by removing the graph transformation rule created in Figure 7.5, since this rule also applies to class C. Figure 7.7 depicts this deletion of the rule in Figure 7.5.

7.2.3 “FORALL” ranges

nonterminal ranges leads to range (COMMA range)*
nonterminal range leads to
  mofType vname (which are both ID’s)
  | tracker
  | patternUse

“FORALL” moftype vname

The following general code snippet of ITL shows one range:

RULE rname formals
FORALL mofType1 vname1

This code can be converted to the graph transformation rule shown in Figure 7.8.

The rule checks if there is a node mofType1, which does not have an incoming edge of the LINK-node of this rule yet. This way this rule will not be executed twice.

When the mofType1 is present, a node LINK is created with an edge to the name of the rule as presented in 7.2.1. Also two edges are created from the LINK node to the mofType1 node. One edge is labelled source to indicate that this mofType is shown in the FORALL part of the tRule. The other edge gets the label of the vname1. This way the variable is captured and it can be traced easily when referred to, later in the tRule.
"FORALL" moftype vname, moftype vname

The following code snippet shows two ranges in the FORALL-part:

```
RULE rname formals
FORALL mofType1 vname1, mofType2 vname2
```

This code is converted to the graph transformation rule shown in Figure 7.9. This rule is converted in the same way as the rule shown in Figure 7.8. This time both mofTypes need to be present for the rule to apply. Both mofTypes cannot have incoming edges from a LINK node. When the rule is applied, new edges are created to both mofTypes.

For two certain mofTypes this rule is executed twice, once with mofType1 as vname1 and once with mofType1 as vname2.

It is not clear whether this rule should be applied twice in ITL. When this would not be desired the negative condition of the incoming edge from the LINK-node to the mofTypes should be named source instead of vname1 and vname2. When this would be done this rule will only be applied once for every couple of mofTypes.
“FORALL” tracker
The following code shows how the tracker is used in a FORALL:

RULE rname formals
FORALL mofType1 vname1, TUSE tname vname

Figure 7.10 shows how this can be converted to a graph transformation.

A node trackingUse has been created with an edge name to a node tname. This makes sure the correct trackingUse is used. The LINK node gets an edge vname towards the trackingUse to bind the variable.

“FORALL” patternUse
The pattern given in the ITL should be merged at syntactical level into the current rule before worrying about graphs. This way there will not remain any patternUse and the tRule can be converted normally.

7.2.4 ( “WHERE” conjunct)?
The following shows what nonterminal conjunct can lead to according to the grammar:

nonterminal conjunct leads to disjunct ("AND" disjunct)*
nonterminal disjunct leads to relation ("OR" relation)*
nonterminal relation leads to
All these possibilities of \textit{conjuncts} will be discussed in the following paragraphs.

\textbf{“WHERE” disjunct “AND” disjunct “AND” disjunct “AND” disjunct}

All \textit{disjuncts} should be created in the graph transformation rule normally. Then only when they all match, the rule will be executed.

\textbf{“WHERE” relation “OR” relation “OR” relation “OR” relation}

First of all the conjunctive normal form of ITL should be rewritten as disjunctive normal form. So when a rule looks like \((\text{relation1 OR relation2) AND (relation3 OR relation4)})\), this should be rewritten as \((\text{relation1 AND relation3) OR (relation1 AND relation4) OR (relation2 AND relation3) OR (relation2 AND relation4)})\). For every part between \textbf{OR}’s a separate graph transformation rule should be created. Of course it should be prevented that more than one of these separate graph transformations are executed. This can be done the same as preventing other rules from being executed more than once. Once a rule is applied it gets a node \textbf{LINK} with an edge to a node with the current rule name. The other rules can have a negative condition which demands that this rule has not been applied yet. The easiest way to make separate graphs is to make separate rules at syntactical level already before creating graph transformation rules, just like for nonterminal \((\text{relatedRules})^*\).

\textbf{“WHERE” “NOT” relation}

The \textit{relation} should be created in the graph transformation rule, where the nodes and edges become a negative application condition. When there are multiple \textbf{NOT’s} following each other, they should be scrapped so that only one or none remains. \textbf{WHERE NOT NOT NOT relation} is the same as \textbf{WHERE NOT relation}. Two \textbf{NOT’s} following each other should be scrapped therefore.

\textbf{“WHERE” patternUse}

The \textit{pattern} given in the ITL should be merged at syntactical level into the current rule before worrying about graphs. This way there will not remain any \textit{patternUse} and the \textit{tRule} can be converted normally.

\textbf{“WHERE” linkOrder}

A \textit{linkOrder} will not be used for this thesis. The reason is that it is assumed that the given meta-models do not contain ordering constraints as told in section 7.1.
"WHERE" factor “ASSIGN” factor

The following code shows how the WHERE clause factor ASSIGN factor is used:

RULE rname formals
FORALL mofType1 vname1, mofType2 vname2
WHERE vname1.feature1 ASSIGN vname2

Figure 7.11 shows how this can be converted to a graph transformation.

```
Figure 7.11 – part of the graph transformation rule for the “factor ASSIGN factor” clause
```

Figure 7.11 shows almost the same graph transformation rule as in Figure 7.9. Only one extra edge is created between mofType1 and mofType2 with as label feature1. This represents that the feature1 of mofType1 should be equal to mofType2 for the rule to be executed.

"WHERE" factor “RELOP” factor

The following code shows how the WHERE clause factor RELOP factor is used:

RULE rname formals
FORALL mofType1 vname1, mofType2 vname2
WHERE vname1.feature1 RELOP vname2

Figure 7.12 shows how this can be converted to a graph transformation.

```
Figure 7.12 looks like Figure 7.11. Except this time there is no edge between mofType1 and mofType2 to represent equality, but a node RELOP has to be matched with as arguments vname1.feature and vname2, which must result to true in order for the rule to apply. This is work that is currently being researched for graph transformations [KAS05]. All instances of all possible combinations with their appropriate result (true or false) are assumed to exist in the source model then. This way every relative operator with pair of integers, strings, etc. can be matched against the source model.
```
7.2.5 “MAKE” targets

nonterminal targets leads to target (COMMA target)*
nonterminal target leads to
    | range
    | assignment
    | patternUse

The following code snippet shows the use of MAKE range of ITL:

RULE rname formals
MAKE mofType2 vname2

This code snippet can be converted to the graph transformation rule shown in Figure 7.13.

A node LINK with an edge to the rule name is created again as shown in section 7.2.1 already. A node mofType2 is created and two edges from the LINK node to the new mofType2 are created. One is called target to represent that this node was created in the
MAKE part. The other is called vname2 which binds the variable name so it can be referenced to in other parts of the tRule.

The following code snippet shows the MAKE range COMMA assignment of ITL:

RULE rname formals
FORALL mofType1 vname1
MAKE mofType2 vname2, vname2.feature2 ASSIGN vname1.feature1

This code is converted to the graph transformation shown in Figure 7.14. As shown in sections 7.2.1 and 7.2.3 already, also here a LINK node is created with edges towards the source mofType. The creation of mofType is done the same way as shown already in Figure 7.13. What is new is the assignation of the feature1 of mofType1 to feature2 of mofType2. Therefore an outgoing edge labelled feature1 from mofType1 is required, which points to a certain node. Then a new edge is created labelled feature2, from node mofType2 which points to the same node as mofType1.feature1.

Figure 7.14 – part of the graph transformation rule for the “MAKE targets” clause

7.2.6 (“LINKING” trackingUses)?

nonterminal trackingUses leads to trackingUse (COMMA trackingUse)*
nonterminal trackingUse leads to
  “LINKING” tname vname
  “WITH” assignment (COMMA assignment)*
  ;

The following code shows the use of trackingUses:

FORALL mofType1 vname1
MAKE mofType2 vname2
LINKING tname3 vname3
WITH vname3.feature1 = vname2 ,
  vname3.feature2 = vname1 ;

This code has been converted to a graph transformation rule which is shown in Figure 7.15.
An edge, with variablename $vname_3$ as its label, from the node $LINK$ is created to a newly created node $trackingUse$. This $trackingUse$ has an outgoing edge to its name $tname_3$. Further a new edge labelled $feature_1$ points to $vname_2$ and a new edge labelled $feature_2$ points to $vname_1$.

![Figure 7.15 – part of the graph transformation rule for the “LINKING trackingUses” clause](image)

### 7.2.7 Factor

nonterminal `factor` leads to

```
| constant |
| path     |
| function |
| vname    |
```

### Constant

nonterminal `constant` leads to

```
| STRING |
| INT    |
```

A constant can be represented by a single node which contains the string or integer value. Figure 7.16 shows a constant integer with value three.

![Figure 7.16 - part of the graph transformation rule for the “constant” clause](image)

### Path

nonterminal `path` leads to

```
vname DOT (feature)+
```

Figure 7.17 shows how to lookup a path in a graph. The $LINK$ node is used to lookup which $mofType$ corresponds to variable $vname$. From this $mofType$ an edge $feature$ leads to the value of the $path$. When there are several features, every feature must be followed to its node, until the final node has been found.
Function

A function can be an operator on one or two integers, like addition. A function can also be an operation on strings, like concatting a number of strings. Figure 7.18 shows how a function that adds two integers can be represented. This is work that is currently being researched for graph transformations [KAS05]. All instances of all possible integer-combinations with their appropriate result are assumed to exist in the source model then. This way every function with pair of integers can be matched against the source model.

Furthermore a function can be an operation on collections, like getting the first element of a collection. Operations on collections have been left out of the scope of this thesis though.

Vname

nonterminal vname leads to

Figure 7.19 shows how to lookup a vname in a graph transformation rule. A LINK node is used to lookup the mofType that corresponds to the variable vname.
7.3 Converting models to instance graphs
This thesis assumes that the model only consists of classes with attributes that have a certain value and binary relations, without ordering constraints, between these classes. All MOF elements presented in section 2.1, including bidirectionality, are supported.

The following points show how to convert the elements in a model to an instance graph:

- Each class can be converted to a node with the name of the class as label. In the example in Figure 6.12 of section 6.3 can be seen that the classes like UMLClass have been converted to nodes with the name of the class as label.

- Each attribute can be converted to an edge with the attributename as its label. This edge is pointed towards a node with the value of the attribute as its label. The source of the attributeedge is the node of the object which contains the attribute. In Figure 6.12 the attribute name is represented as an edge from its UMLClass object towards its value Person, represented as a node.

- A relation between two classes is converted to two edges, pointing in inverse directions and having the rolename as label. When two rolenames on two different edges between two certain class-nodes happen to be the same, they should be renamed to the same value mentioned already for the conversion of the metamodel in section 7.1. The example in Figure 6.12 shows the relation between UMLClass and UMLAttribute with two edges pointing in inverse directions labelled as owner and feature.

For models multiplicity is not applicable at all, so multiplicity does not have to be converted.

7.4 Applying graph transformation
This step should be done completely by a graph transformation tool. When the graph transformation rules created in section 7.2 are given to a graph transformation tool along with the instance graph created in section 7.3, a combined instance graph of target and source should be generated.

7.5 Extracting target instance graphs out of generated combined instance graphs
When the rules mentioned in section 7.2 are applied, a model with LINK nodes, trackingUses, source model and target model is generated. In order to extract the target model out of this combined model, a simple method would be to delete all LINK nodes with the source node it points to and to delete all trackingUses. This method fails though when not all elements in the source instance graph are used as source in one of the transformation rules.

That’s why a few steps of rules need to be applied. These steps are categorized in three steps, which will be explained here.
7.5.1 Step 1
The first steps consist of tagging all target nodes with a node `KEEPTARGET`. This is done by first applying the graph transformation rule of Figure 7.20. This rule adds a tag node `KEEPTARGET` to the target node of a `LINK` node. This is only done when the node does not already contain a tag node.

![Figure 7.20 – graph transformation rule marking target node](image)

The graph transformation rule of Figure 7.21 will mark all other nodes that can be reached somehow from the target edge of the `LINK` node, which has been marked by the graph transformation rule of Figure 7.20. Even if there are hundred nodes in between, this graph transformation rule will tag the nodes if they can be reached from the target edge of the `LINK` node.

![Figure 7.21 – graph transformation rule marking all target nodes](image)

When these rules have been applied, all nodes that belong to the target instance graph have been tagged.
7.5.2 Step 2
Now it is time to delete all nodes that have not been marked by Step 1. These would be all source nodes, trackingUses and LINK nodes. The deletion is performed by the graph transformation rule of Figure 7.22.

![Figure 7.22 - graph transformation rule deleting unmarked nodes](image)

When this step has been applied the graph only contains the nodes of the target instance graph and the tag nodes.

7.5.3 Step 3
The temporary tag nodes can be deleted now. This is done by the graph transformation rule of Figure 7.23.

![Figure 7.23 – graph transformation rule deleting temporary mark nodes](image)

The instance graph now only contains the target nodes and represents the target instance graph.

7.6 Converting target instance graphs to target models
Converting target instance graphs to target models is basically the inverse process of the conversion explained in section 7.3. All info needed is present in the target instance graph as created by previous steps. The following conversion steps are needed:

- All nodes in the target instance graph, which have a name which corresponds with a name of a class in the target meta-model, need to be converted to an object.
- All pairs of edges between these nodes need to be converted to an association, with two role names, between the two generated objects. The target meta-model is needed to distinguish which role names belong together.
- All remaining edges need to be converted to variable names of the object, whose node is the source of the edge. The target nodes of these edges are the values of these variables.
7.7 Conclusion

This chapter gave a general approach for all key parts of conversion from ITL to graph transformations.
Section 7.1 discussed the conversion from source and target meta-models to type graphs.
In section 7.2 the translation from the ‘A To B transformation definition’ of ITL to ‘Graph transformation rules A To B’ was discussed.
Section 7.3 showed how to convert a model to an instance graph.
Section 7.4 told how to apply the graph transformation to get the combined instance graph.
Section 7.5 showed how to extract the target instance graph from this combined instance graph.
Finally section 7.6 showed the conversion of an instance graph to the corresponding model.
8 QVT2Graph tool

This chapter shows the design and explanation of a tool that has been created for this thesis. The goal of this prototype tool is to apply the concepts devised in chapter 7 and to proof these concepts in practice this way.

8.1 Design

Figure 8.1 shows the same diagram as Figure 4.4, but this time concrete ways to implement the conversions are shown.

- The meta-models and model should be given in XMI format [OMG02]. XMI can be exported by tools, like Borland Together [BOR], which can design class diagrams.

- The given meta-models and models in XMI format can be read by a DOM parser which constructs a tree and iterates over tags. This way the files can be converted to type graphs and instance graphs.

- The given A to B transformation definition can be scanned and parsed by ANTLR, a parser generator for Java. Using ANTLR, each nonterminal can be converted to graph transformation rules as explained in section 7.2. The target meta-model is also needed in this process in order to support bidirectional links.

- Using the graph transformation rules and the instance graph of the source model, a graph transformation can be applied, using GROOVE (Graph-based Object-Oriented Verification), which gives a combined instance graph with backtracking information of the source and target model. GROOVE is a tool which can edit and model graphs. It can also automatically perform graph transformations and generate the final states and all intermediate states. [REN03]

- The instance graph of the target model has to be extracted from the combined instance graph by applying the graph transformation rules explained in section 7.5 to the combined instance graph, using GROOVE.

- This target instance graph is converted to a target model using a DOM writer to write the model as XMI file. This XMI file can be imported again by classdiagramtools like Borland Together.
Figure 8.2 shows a simplified class diagram for the classes of the tool. A main class calls the ANTLR classes which parse the transformation definition. The ANTLR parser calls the GROOVE classes to create graphs. The main class also calls classes which convert models and meta-models to instance graphs and type graphs respectively. The class which converts the instance graphs to models is also called by the main class. All these
classes use the GROOVE library to create and read graphs and they use a DOM parser to parse and write XMI files.

![Figure 8.2 – simplified class diagram of classes of QVT2Graph tool](image)

### 8.2 Description

The QVT2Graphs tool consists of four steps. This section describes what these steps do.

#### 8.2.1 Step 1

Step one actually consists of several tasks. Since these tasks can be performed independently of each other, they have been grouped into one step for this prototype. Figure 8.3 shows a screenshot of step one. All tasks will be discussed shortly.
Source meta-model

The source meta-model should be made in the application Borland Together 6.1 (other programs and versions might work, but have not been tested for this prototype). In Borland Together, a new project should be created with as default language Java. In the project a new Class diagram should be created. In this class diagram Classes with attributes and relations between classes and inheritance between classes can be constructed to represent the source meta-model.

When the meta-model is finished, it should be exported as a XMI file. In the prototype only XMI 1.1 for UML 1.4 (OMG) is supported. The encoding should be ASCII. Figure 8.4 shows the “Export to XMI” dialog of Borland Together 6.1.

The exported XMI file can be chosen in the QVT2Graphs tool as source meta-model in step one.
**Target meta-model**
What has been done for the source meta-model should also be done for the target meta-model.

**Transformation rules**
A text file should be created which contains the transformation rules which conform to the adjusted ITL (the IBM Transformation Language) grammar shown in Appendix B. This text file can be chosen in step one as transformation rules.

**Source model**
The source model should be created also in Borland Together 6.1. The project which contains the source meta-model should be opened and a new class diagram should be created. In this new class diagram, objects can be created which are instantiates of classes in the source meta-model. Attributes can be given values and associations between objects can be made. When the object model is finished, it should also be exported as XMI file just like the Source meta-model.

**Output folder**
An output folder should be chosen where all created models and files will be stored.

**Convert**
When clicking the convert button there will be four conversions.
- The source meta-model will be converted into a source type graph
- The target meta-model will be converted into a target type graph
- The ITL rules will be converted into graph transformation rules
- The source model will be converted into a source instance graph

8.2.2 step 2
Figure 8.5 shows a screenshot of step two after the conversion of step one.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source typegraph</td>
<td>example\output\uml\metamodel.gst</td>
<td>View..</td>
<td></td>
</tr>
<tr>
<td>target typegraph</td>
<td>example\output\javametamodel.gst</td>
<td>View..</td>
<td></td>
</tr>
<tr>
<td>graph rule folder</td>
<td>example\output\uml2\java.gps</td>
<td>View..</td>
<td></td>
</tr>
<tr>
<td>source instance graph</td>
<td>example\output\uml2\java.gps\start.gst</td>
<td>View..</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.5 – screenshot step two of tool**
In step two all models and graph transformation rules, created in step one, can be viewed by pressing the view buttons. When pressing the button “Perform transformation”, the graph transformation rules will be applied to the source instance graph in order to create a combined instance graph.

8.2.3 Step 3

Figure 8.6 shows a screenshot of step three.

In step three the combined instance graph, created by performing the transformation in step two, can be viewed by pressing the view button.

By pressing the extract button, the target instance graph will be extracted from the combined instance graph. The target instance graph will be created in the output folder chosen in step one with the filename chosen in step three.

8.2.4 Step 4

Figure 8.7 shows a screenshot of step four.

In step four the target instance graph, created in step three, can be viewed by pressing the view button. By pressing the button “Convert to target model” a XMI file will be created of the target model. This XMI file can be imported using Borland Together 6.1.
8.2.5 Log window

Figure 8.8 shows a screenshot of the log window.

The log window shows the progress of the steps, which files have been created, warnings and errors, etc.

8.3 Example UML to Relational database model

A second example has been created for this thesis and it has only been executed by the tool that has been presented in this chapter. The example has been based on examples of a tool Tefkat [DSTC05], and on the QVT submission of QVT Partners [QVT03]. The transformation being executed is from UML to a relational database model.

The example input source model is presented in Figure 8.9. This represents an association between two UML classes. One class represents the Company and the other the Owner of the Company. The Company class has an attribute companyName, which has a StringBuffer as primitive datatype.

The type graphs and graph transformation rules have been created with the tool and can be found in Appendix.

Figure 8.10 shows the resulting relational target model, which the QVT2Graph tool produced. The classes have been converted to tables with keys. The attributes are represented in columns now and a foreign key has been created for the association between the two classes. The foreign key has a column in the table of one class, and this key points to the other class.
8.4 Conclusion

This chapter showed the design and explanation of a tool that has been created for this thesis. Not everything has been implemented for this prototype. This because the subjects that have been left out are straightforward and do not help to prove the main points of this thesis. The following points have not been implemented:

- **disjuncts, patternUses, extends, supersedes, OR’s and when subclasses exist of a class**: in all these cases rules have to be split into multiple rules at syntactical level. This would be done by a preprocessor that would parse the ITL definition and has as output more rules of ITL. This would have to be done before worrying about graphs and can only be done manually currently.
- **Not all kinds of XMI files can be parsed by the tool.** There is quite some amount of different XMI/UML/MOF file formats. It would not help to prove the main points of this thesis to be able to read more than one XMI file format.
Conclusions and recommendations

The aim of this thesis was to take the QVT proposal by IBM and partners [DSTC04] and to be able to represent any arbitrary transformation definition as a graph production system.

Chapter 2 discussed the relevant context of MDA, QVT, MOF, graph transformations and ITL. After this, chapter 3 classified both ITL and the graph production system, using the classification system of Czarnecki [CZA03], and compared the features of both in order to see how well a graph production system can behave the same as ITL. Chapter 4 then analyzed the problem studied in this thesis and described what steps needed to be taken in order to solve this. Chapter 5 showed that ITL isn’t very clear and that the grammar and examples were not correct and that the grammar had to be adjusted in order to have a working grammar, which can be seen in Appendix B. Chapter 6 then showed an adjusted example of ITL (the IBM Transformation Language). In this example the meta-models were converted to type graphs, the rules were converted to graph transformation rules and the source model was converted to an instance graph. Then the graph transformation rules were applied to the instance graph to create a combined instance graph. It was then shown how to extract the target instance graph out of this combined instance graph and finally how to convert this to a target model.

Chapter 7 then showed a general approach that is able to perform all the conversion steps of chapter 6 for any arbitrary example. LinkOrders have not been handled for this thesis, because the meta-models are assumed not to contain ordering constraints. To prove that the general approach works, a tool prototype was developed in chapter 8. It showed that the assumptions in chapter 7 worked in practice. All steps of the example transformation and general transformations have been implemented and are working. Trivial steps like syntactical conversion were left out of the tool, since this should be straightforward and does not prove the main goals of this thesis.

The intentions of IBM have been kept as far as it was clear. The output with graphs is the same as it would be with ITL. Challenges for graph production systems, like bidirectional associations and traceability have been implemented. It has been managed that rules are only executed once on a certain source element. Furthermore variable names and all backtracking information are left intact in the combined instance graph.

This thesis showed in practice that graph transformations are a good well-established solution for model transformations. Many people have talked about it [GRU05, KUS04], but no real implementation has been created so far for MDA model transformation, of any arbitrary source and target model, by using graph production systems.

If the work of this thesis will be continued, the following points are recommended which are still open:

- Ordering constraints could be assumed in the meta-model, and because of that LinkOrders would have to be handled also.
- When development of the tool is continued, the syntactical conversion could be implemented. Also it could be implemented that any XMI version can be read and written by the tool.
• Instead of using syntactical conversion for patternUses, extending other rules and for having separate rules for any subclasses in the meta-model, a more sophisticated solution could be created. For subclasses in the meta-model, there could be a matching node which matches all the subclasses with an OR, e.g. a node with a selfedge UMLClassifier | UMLClass | UMLInterface. This has not been tested though.
References

  http://www.omg.org/docs/ad/03-03-25.pdf
  http://www.omg.org/docs/ad/03-08-05.pdf
- [BOR] Borland – Borland Together tool  
  http://www.omg.org/docs/ad/03-08-07.pdf
- [DSTC05] DSTC QVT team – Tefkat tutorial  
  http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/
  http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/GGZEGV05.pdf
  http://www.omg.org/docs/ad/03-08-11.pdf
  http://www.cs.kent.ac.uk/projects/ocl/oclmdewsuml04/papers/6-kuster_sendall_wahler.pdf
• [OMG02b] OMG – *Request For Proposal: MOF 2.0 Query / Views / Transformations*, April 2002
• [OMG03] OMG - *Meta Object Facility (MOF) specification*, version 1.4, April 2002
  http://www.omg.org/docs/formal/02-04-03.pdf
• [QVT03] QVT-partners – *Revised submission for MOF 2.0 Query / Views / Transformations RFP*, version 1.1 August 18th, 2003
  http://www.omg.org/docs/ad/03-08-08.pdf
• [RENS05] Rensink, A., University of Twente, Netherlands – *The Joys of Graph Transformation*, Newsletter of Dutch Association for Theoretical Computer Science, February 9, 2005
• [UML03] OMG – *Unified Modeling Language, formal specification* v1.5 2003
  http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf
Appendices

Appendix A contains the given example in ITL.
Appendix B contains the original and the modified concrete syntax grammar that is used in this thesis.
Appendix C contains a second example transformation. The example shows a transformation from a UML model to a relational model.
A. Appendix A

The following examples are taken from chapter 3 of ITL (the IBM Transformation Language):

**Figure 3.4**
The rule UmlClassifierToJavaClass converts a UmlClassifier to a JavaClass.

TRANSFORMATION Uml2Java(uml, java, tagging)

TRACK USING TModel{
KEY OF JavaClassFromUMLClassifier IS umlClassifier;
...}

RULE UmlClassifierToJavaClass(uc,jc)
FORALL UMLClassifier uc
MAKE JavaClass jc,
jc.name = uc.name
LINKING JavaClassFromUMLClassifier jcuc
WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;

**Figure 3.5**
The rule UmlAttributeToJavaField converts a UMLAttribute to a JavaField.

RULE UmlAttributeToJavaField(ua,jf)
FORALL UMLAttribute ua
WHERE JavaClassFromUMLClassifier jcuc AND
jcuc.umlClassifier = ua.owner AND
JavaClass jc = jcuc.javaClass
MAKE JavaField jf,
jf.owner = jc
LINKING FieldFromAttr ffa
WITH ffa.javaField = jf, ffa.umlAttr = ua;

**Figure 3.6**
The rule CopyrightToJavaClass converts a tagging parameter to a copyright statement in the Java model.

RULE CopyrightToJavaClass
FORALL Tag@tagging t
WHERE t.name = 'Copyright'
AND t.type = 'UMLClassifier'
AND JavaClassFromUMLClassifier jcuc
AND JavaClass jc = jcuc.javaClass
MAKE jc.comment = t.value;

**Figure 3.7**
The rule UmlClassifierToJavaClass converts a UmlClassifier to a JavaClass. Patterns are used to simplify the code of the rule.
DEFINE PATTERN UmlClassifierAndName(uc,name)
FORALL UMLClassifier uc
WHERE uc.name = name;

DEFINE PATTERN JavaClassAndName(jc,name)
MAKE JavaClass jc AND
jc.name = name;
RULE UmlClassifierToJavaClass(uc,jc)
FORALL UMLClassifierAndName(uc, n)
MAKE JavaClassAndName(jc, n)
LINKING JavaClassFromUMLClassifier jcuc
WITH jcuc.javaClass = jc, jcuc.umlClassifier = uc;

Figure 3.8
The rule UmlInterfaceToJavaInterface converts a UMLInterface to a JavaInterface.

RULE UmlInterfaceToJavaInterface(uc,jc)
SUPERSEDES umlClassifierToJavaClass(uc,jc)
FORALL UMLInterface uc
MAKE JavaInterface jc,
jc.name = uc.name
LINKING JavaIntfFromUMLIntf jiui
WITH jiui.javaIntf = jc, jiui.umlIntf = uc;

The rule UmlClassToJavaClass converts a UmlClass to a JavaClass with constructor.

RULE UmlClassToJavaClass(uc,jc)
EXTENDS UmlClassifierToJavaClass(uc,jc)
MAKE JavaMethod m,
m.name = uc.name,
jc.constructor = m
LINKING JavaConsFromUMLClass consFromClass
WITH
consFromClass.constructor = m,
consFromClass.umlClass = uc;

Figure 3.9
The rule umlParameterToJavaParameter converts a UMLParameter to a JavaParameter.

RULE umlParameterToJavaParameter
FORALL UMLParameter up
MAKE JavaParameter jp,
up.name = jp.name
LINKING JavaParamFromUMLParam jpup
WITH jpup.javaParam = jp, jpup.umlParam = up;

The rule parameterOrdering keeps the order of the JavaParameters the same as the UMLParameters.

RULE parameterOrdering
FORALL UMLOperation uOp, UMLParameter up1, UMLParameter up2
WHERE up1 BEFORE up2 IN uOp.parameter
AND JavaMethodFromUML jmu
AND jmu.umlOp = uOp
AND Method m = jmu.javaMethod
AND JavaParamFromUMLParam jpup
AND jpup.umlParam = up1
AND JavaParameter jp1 = jpup.javaParam
AND JavaParamFromUMLParam jpup2
AND jpup2.umlParam = up2
AND JavaParameter jp2 = jpup javaParam
MAKE jp1 BEFORE jp2 IN m.arg;
}
# B. Appendix B

This appendix shows the concrete syntax grammar of ITL (the IBM Transformation Language). The left column contains the original grammar as presented in ITL [DSTC04]. The right column shows the grammar that has been modified for this thesis because the original of ITL was not correct. Chapter 5 explains the encountered problems in the original grammar of ITL.

<table>
<thead>
<tr>
<th>Original</th>
<th>Modified for this thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transformation : &quot;TRANSFORMATION&quot; ID formals body ;</code></td>
<td><code>transformation : &quot;TRANSFORMATION&quot; ID formals body ;</code></td>
</tr>
<tr>
<td><code>formals : LBRACK ( vardecls )? RBRACK ;</code></td>
<td><code>formals : LBRACK ( vardecls )? RBRACK ;</code></td>
</tr>
<tr>
<td>`body : (trackingImport</td>
<td>patternDefn</td>
</tr>
<tr>
<td>`vardecl : ( &quot;SOURCE&quot;</td>
<td>&quot;TARGET&quot; ) ID ;`</td>
</tr>
<tr>
<td><code>vardecls : vardecl ( COMMA vardecl )* ;</code></td>
<td><code>vardecls : vardecl ( COMMA vardecl )* ;</code></td>
</tr>
<tr>
<td><code>trackingImport : &quot;TRACK&quot; USING ID L CURLY trackbody R CURLY SEMI ;</code></td>
<td><code>trackingImport : &quot;TRACK&quot; USING ID L CURLY trackbody R CURLY SEMI ;</code></td>
</tr>
<tr>
<td><code>trackbody : (&quot;KEY&quot; &quot;OF&quot; cname &quot;IS&quot; feature ( COMMA feature ) SEMI ) * ;</code></td>
<td><code>trackbody : (&quot;KEY&quot; &quot;OF&quot; cname &quot;IS&quot; feature ( COMMA feature ) SEMI ) * ;</code></td>
</tr>
<tr>
<td>`patternDefn : &quot;PATTERN&quot; pname formals ( &quot;FORALL&quot; ranges ( &quot;WHERE&quot; conjunct )?</td>
<td>`patternDefn : &quot;DEFINE PATTERN&quot; pname formals ( &quot;FORALL&quot; ranges ( &quot;WHERE&quot; conjunct )?</td>
</tr>
<tr>
<td></td>
<td>&quot;MAKE&quot; targets ) SEMI ;`</td>
</tr>
<tr>
<td>`true : &quot;RULE&quot; rname formals ( relatedRules ) * ;&quot;FORALL&quot; ranges ( &quot;WHERE&quot; conjunct )?</td>
<td>`true : &quot;RULE&quot; rname formals ( relatedRules ) * ;&quot;FORALL&quot; ranges ( &quot;WHERE&quot; conjunct )?</td>
</tr>
<tr>
<td></td>
<td>&quot;MAKE&quot; targets ( &quot;LINKING&quot; trackingUses )? SEMI ;`</td>
</tr>
</tbody>
</table>
;  

tname : ID  

;  

pname : ID  

;  

ranges : range ( COMMA range )*  

;  

conjunct : disjunct ( "AND" disjunct )*  

;  

rname : ID  

;  

cname : ID  

;  

relatedRules : ( "extends" |
"supersedes" |
"overrides" )  

rname formals ( "," rname formals )*  

;  

targets : target ( COMMA target )*  

;  

trackingUses : trackingUse ( COMMA trackingUse )*  

;  

range : mofType vname  

;  

mofType : ID  

;  

vname : ID  

;  

disjunct : relation ( "OR" relation )*  

;  

relation : LBRACK conjunct RBRACK  

| "NOT" relation

| patternUse 

| tracker

;  

range : mofType vname

;
:fname actuals
;

feature
:ID
;

ID : (a...z | A...Z | 0...9 )+
;
C. Appendix C

This appendix contains a second example which shows the results, given by the QVT2Graph tool made for this thesis. This example converts UML models to relational database models. The example has been based on examples of a tool Tefkat [DSTC05], and on the QVT submission of QVT Partners [QVT03].

Meta-model to type graph

This paragraph converts the meta-models of UML and the relational database to type graphs.

UML

Figure C.1 shows a simplified UML meta-model.

![Figure C.1 – meta-model UML](image)

The meta-model of UML is converted to a type graph of UML. The resulting type graph can be seen in Figure C.2.
RDBMS

Figure C.3 shows a simplified relational meta-model.
The meta-model of the relational model is also converted to its type graph. The resulting type graph can be seen in Figure C.4.
**Rules to graph transformation rules**

The following rules have been converted to graph transformation rules.

**Class2Table**

The rule **Class2Table** converts UML classes, with the persistent property, to a Table and a Key. The name of the class is assigned to the table and the key. Furthermore a relation between the Table and the Key is made.

RULE Class2Table(c, t, k)
FORALL UMLClass c
WHERE c.kind = "persistent"
MAKE Table t, Key k,
t.name = c.name, t.key = k, k.name = c.name
LINKING TableAndKeyFROMUMLClass tkuc
WITH tkuc.umlClass = c, tkuc.table = t, tkuc.key = k;

This rule has been converted to the graph transformation rule of Figure C.5.

![Figure C.5 – graph transformation rule Class2Table](image)

**Attr2Column**

The rule **Attr2Column** converts attributes to columns and creates the link with the corresponding table. It also assigns the name of the Attribute to the Column.

RULE Attr2Column(c, a, t, k, col)
FORALL Attribute a, TUSE TableAndKeyFROMUMLClass tkuc
WHERE a.owner = tkuc.umlClass
MAKE Column col, col.name = a.name,
This rule is converted to the graph transformation rule of Figure C.6.

![Figure C.6 – graph transformation rule Attr2Column](image)

**ForeignKeyForAssoc**

The rule `ForeignKeyForAssoc` converts an association to a foreign key. This rule also creates a column which contains the key towards the other table.

```
RULE ForeignKeyForAssoc (assoc)
FOR ALL Association assoc, TUSE TableAndKeyFROMUMLClass tkuc1, TUSE TableAndKeyFROMUMLClass tkuc2
WHERE tkuc1.umlClass = assoc.source AND Table t = tkuc1.table AND
tkuc2.umlClass = assoc.target AND Key k = tkuc2.key
MAKE Column col, ForeignKey fk, col.name = assoc.name, col.table = t, fk.table
= t, fk.column = col, fk.refersTo = k;
```

This rule is converted to the graph transformation rule of Figure C.7
Source model to instance graph

The example input source model is presented in Figure C.8. This represents an association between two UML classes. One class represents the Company and the other the Owner of the Company. The Company class has an attribute companyName, which has a StringBuffer as primitive datatype.

This source model has been converted to the type graph in Figure C.9.
Applying graph transformation

When the graph transformation rules are applied to created UML instance graph of the previous section, a combined instance graph is generated. This combined instance graph can be seen in Figure C.10. This instance graph is too large to be displayed in a presentable manner.
Extracting target instance graph out of combined instance graph

When all the LINK nodes and the trackingUses and the source model are deleted out of the combined instance graph of Figure C.10, the model in Figure C.11 is obtained.
Converting target instance graph to target model

The extracted target instance graph in Figure C.11 is converted to a target model. The converted target model can be seen in Figure C.12.

The classes have been converted to tables with keys. The attributes are represented in columns now and a foreign key has been created for the association between the two classes. The foreign key has a column in the table of one class, and this key points to the other class.