APPLYING METAMODELS IN SOFTWARE MODEL TRANSFORMATION METHODS

PhD Thesis

Tihamér Levendovszky

Advisor:
Dr. Hassan Charaf Ph.D.
Associate Professor

Budapest University of Technology and Economics
Department of Automation and Applied Informatics

Budapest, 2005
# Table of Contents

1. **INTRODUCTION** ............................................................................................................... 2  
   1.1 Thesis Contributions ....................................................................................................... 2  
   1.2 Thesis Structure ............................................................................................................. 4  

2. **MOTIVATION** .................................................................................................................. 5  
   2.1 Models and Metamodels ................................................................................................. 5  
      2.1.1 The Unified Modeling Language ........................................................................... 5  
      2.1.2 Instantiation .......................................................................................................... 6  
      2.1.3 Metamodelling and UML ....................................................................................... 9  
   2.2 The Claim for Software Model Transformation ............................................................. 11  
      2.2.1 Model-Driven Architecture ................................................................................. 11  
      2.2.2 Transformations in CASE Tools .......................................................................... 12  
      2.2.3 Transformations and Model Testing ................................................................. 13  
   2.3 Model Transformation Techniques ................................................................................ 14  
      2.3.1 Model Transformation and MDA ......................................................................... 15  
      2.3.2 The Traversal Approach ...................................................................................... 15  
      2.3.3 The Visual Approach ............................................................................................ 16  
   2.4 Open Issues .................................................................................................................. 17  
   2.5 Chapter Summary ......................................................................................................... 17  

3. **METAMODELING AND MODEL TRANSFORMATION SYSTEMS** .......................... 19  
   3.1 Metamodeling Tools ..................................................................................................... 19  
   3.2 Model Integrated Computing ........................................................................................ 20  
   3.3 Model Transformation Systems .................................................................................... 21  
      3.3.1 PROGRES .............................................................................................................. 21  
      3.3.2 GReAT .................................................................................................................. 22  
      3.3.3 The VIATRA System .............................................................................................. 24  
      3.3.4 Other Approaches ................................................................................................. 25  
   3.4 The Visual Modeling and Transformation System ......................................................... 27  
      3.4.1 Design Objectives ................................................................................................. 27  
      3.4.2 Overview .............................................................................................................. 29  
      3.4.3 An n-layer metamodelling environment ............................................................. 30  
      3.4.4 Model Transformations ......................................................................................... 33  
   3.5 Chapter Summary ......................................................................................................... 38  
      3.5.1 Related Work ........................................................................................................ 38  
      3.5.2 VMTS Summary ................................................................................................. 38  
      3.5.3 Open Issues in VMTS ......................................................................................... 39  

4. **THE ALGORITHMIC BACKGROUND** ........................................................................ 40  
   4.1 Pattern Matching and Subgraph Isomorphism ............................................................... 40  
      4.1.1 Ullmann’s Algorithm ......................................................................................... 40  
      4.1.2 The VF2 Algorithm ......................................................................................... 43  
   4.2 Properties of Instantiation ............................................................................................. 45  
   4.3 Instantiating the Model Structure ................................................................................ 47  
   4.4 Type-Aware Matching with Heuristics .......................................................................... 52  
   4.5 A Metamodel-Based Matching Algorithm .................................................................... 54  
   4.6 Chapter Summary ......................................................................................................... 58  

5. **METAMODELING AND HOMPOMORPHIC MAPPINGS** .................................... 59  
   5.1 Mappings ..................................................................................................................... 59  
   5.2 The Homomorphic Typeof Mapping .......................................................................... 60  
   5.3 The Homomorphic Instantiation .................................................................................. 64  
   5.4 Chapter Summary ......................................................................................................... 66  

6. **PARALLEL RULE EXECUTION** .................................................................................... 67
# Table of Contents

6.1 Category Theory ............................................................................................................. 67
6.2 Graph Transformation .................................................................................................... 72
  6.2.1 The DPO Approach ................................................................................................. 73
  6.2.2 Other Results .......................................................................................................... 77
6.3 Extending the DPO Approach to Metamodel-Based Rewriting Rules .................. 78
6.4 Chapter Summary ........................................................................................................... 84

7. Evaluation ....................................................................................................................... 85

8. Application of the Results ............................................................................................. 86
  8.1 Model-Driven Architecture .......................................................................................... 86
    8.1.1 Code Generation for Statecharts with VMTS ......................................................... 87
    8.1.2 Code Generation for UML Class Diagram ............................................................ 92
  8.2 Generative Programming .............................................................................................. 94
    8.2.1 Normalizing feature models .................................................................................. 95
Chapter Summary .................................................................................................................... 99

9. Conclusions .................................................................................................................... 100
  9.1 Summary .................................................................................................................... 100
  9.2 Future Work ................................................................................................................ 102

10. Appendix .......................................................................................................................... 104
  A. Detailed Proofs ............................................................................................................. 104
    Proof for Proposition 6.2 ............................................................................................... 104
    Proof for Proposition 6.3 ............................................................................................... 105
  B. Comparison Table for Model Transformation Tools .................................................. 107

Bibliography ......................................................................................................................... 108
List of Tables and Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 2.1</td>
<td>METAMODEL (A) AND ITS CORRESPONDING MODEL (B)</td>
<td>7</td>
</tr>
<tr>
<td>FIGURE 2.2</td>
<td>THE UML STATECHART DIAGRAM METAMODEL AND A SAMPLE STATECHART MODEL</td>
<td>8</td>
</tr>
<tr>
<td>FIGURE 2.3</td>
<td>A METAMODEL FOR UML STATECHART DIAGRAM</td>
<td>8</td>
</tr>
<tr>
<td>FIGURE 2.4</td>
<td>A SAMPLE UML STATECHART DIAGRAM</td>
<td>9</td>
</tr>
<tr>
<td>FIGURE 2.5</td>
<td>THE MOF METAMODELING PYRAMID</td>
<td>9</td>
</tr>
<tr>
<td>FIGURE 2.6</td>
<td>THE MOST PREVALENT PATHS OF SOFTWARE MODEL TRANSFORMATIONS</td>
<td>11</td>
</tr>
<tr>
<td>FIGURE 2.7</td>
<td>ASM REFINEMENT</td>
<td>13</td>
</tr>
<tr>
<td>FIGURE 3.1</td>
<td>THE GME METAMODEL (SUBSET)</td>
<td>19</td>
</tr>
<tr>
<td>FIGURE 3.2</td>
<td>THE EMF METAMODEL</td>
<td>20</td>
</tr>
<tr>
<td>FIGURE 3.3</td>
<td>THE BASIC CONCEPTS OF MIC</td>
<td>22</td>
</tr>
<tr>
<td>FIGURE 3.4</td>
<td>THE OVERALL ARCHITECTURE OF GREAT</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 3.5</td>
<td>AN EXAMPLE FOR THE GREAT TRANSFORMATION LANGUAGE</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 3.6</td>
<td>THE GROUPING CONSTRUCT IN THE GREAT PATTERN LANGUAGE</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 3.7</td>
<td>THE METAMODEL OF THE GREAT CONTROL STRUCTURE</td>
<td>24</td>
</tr>
<tr>
<td>FIGURE 3.8</td>
<td>VIATRA2 USAGE CONCEPTS</td>
<td>25</td>
</tr>
<tr>
<td>FIGURE 3.9</td>
<td>THE VMTS SYSTEM ARCHITECTURE</td>
<td>30</td>
</tr>
<tr>
<td>FIGURE 3.10</td>
<td>THE READ-ONLY ROOT METAMODEL OF AGSI</td>
<td>31</td>
</tr>
<tr>
<td>FIGURE 3.11</td>
<td>THE ATTRIBUTE INSTANTIATION CHAIN</td>
<td>32</td>
</tr>
<tr>
<td>FIGURE 3.12</td>
<td>RULE SPECIFICATION IN AGSI</td>
<td>36</td>
</tr>
<tr>
<td>TABLE 3.1</td>
<td>A COMPARISON OF METAMODELING TOOLS</td>
<td>38</td>
</tr>
<tr>
<td>FIGURE 4.1</td>
<td>(A) PERmuting rows. (B) PERmuting the corresponding columns. (C) comparing the result to $A_{LHS}^{LHS}$</td>
<td>41</td>
</tr>
<tr>
<td>FIGURE 4.2</td>
<td>A SIMPLE MATCH ALGORITHM</td>
<td>41</td>
</tr>
<tr>
<td>FIGURE 4.3</td>
<td>ULLMANN’S ALGORITHM</td>
<td>42</td>
</tr>
<tr>
<td>FIGURE 4.4</td>
<td>THE OUTLINE OF THE VF2 ALGORITHM</td>
<td>43</td>
</tr>
<tr>
<td>FIGURE 4.5</td>
<td>A SIMPLE FEASIBLE PROCEDURE</td>
<td>44</td>
</tr>
<tr>
<td>FIGURE 4.6</td>
<td>THE FEASIBILITY CHECK OF THE VF2 ALGORITHM</td>
<td>45</td>
</tr>
<tr>
<td>FIGURE 4.7</td>
<td>COMPUTING THE CANDIDATES IN THE VF2 ALGORITHM</td>
<td>45</td>
</tr>
<tr>
<td>FIGURE 4.8</td>
<td>A GENERAL BIDIRECTIONAL ASSOCIATION WITH NONZERO INTEGER MULTIPLICITIES</td>
<td>45</td>
</tr>
<tr>
<td>FIGURE 4.9</td>
<td>PARALLEL LINKS</td>
<td>46</td>
</tr>
<tr>
<td>FIGURE 4.10</td>
<td>AN INFINITE CONSTRUCT FOR A CLASS DIAGRAM WHICH CANNOT BE ANALYZED BY BLOCKS</td>
<td>48</td>
</tr>
<tr>
<td>FIGURE 4.11</td>
<td>THE CREATEIMM ALGORITHM</td>
<td>48</td>
</tr>
<tr>
<td>FIGURE 4.12</td>
<td>AN EXAMPLE FOR CREATING IMM</td>
<td>49</td>
</tr>
<tr>
<td>FIGURE 4.13</td>
<td>THE IMM ELIMINATION ALGORITHM</td>
<td>49</td>
</tr>
<tr>
<td>FIGURE 4.14</td>
<td>AN UML CLASS DIAGRAM HAVING NO VALID INSTANCE</td>
<td>50</td>
</tr>
<tr>
<td>FIGURE 4.15</td>
<td>A SIMPLE FEASIBLE PROCEDURE</td>
<td>52</td>
</tr>
<tr>
<td>FIGURE 4.16</td>
<td>COMPUTING CANDIDATES INMetaVF2 ALGORITHM</td>
<td>53</td>
</tr>
<tr>
<td>FIGURE 4.17</td>
<td>A SIMPLE COUNTEREXAME. (A) HOST GRAPH, (B) LHS</td>
<td>54</td>
</tr>
<tr>
<td>FIGURE 4.18</td>
<td>TWO INSTANTIATION POSSIBILITIES</td>
<td>54</td>
</tr>
<tr>
<td>FIGURE 4.19</td>
<td>An UML-BASED PATTERN MATCHING ALGORITHM</td>
<td>56</td>
</tr>
<tr>
<td>FIGURE 4.20</td>
<td>(A) METAMODEL, (B) INSTANCE MODEL OF THE PRECEDENCE LIST EXAMPLE</td>
<td>56</td>
</tr>
<tr>
<td>TABLE 4.1</td>
<td>THE MATCHING PROCESS</td>
<td>57</td>
</tr>
<tr>
<td>FIGURE 5.1</td>
<td>INSTANTIATION</td>
<td>61</td>
</tr>
<tr>
<td>TABLE 5.1</td>
<td>NOTATIONS</td>
<td>64</td>
</tr>
<tr>
<td>FIGURE 6.1</td>
<td>COPRODUCT (SUM)</td>
<td>69</td>
</tr>
<tr>
<td>FIGURE 6.2</td>
<td>A SAMPLE FOR QUOTIENT SETS OF GRAPHS</td>
<td>71</td>
</tr>
<tr>
<td>FIGURE 6.4</td>
<td>THE QUOTIENT SET OF $G_{1}UG_{2}$</td>
<td>71</td>
</tr>
<tr>
<td>FIGURE 6.5</td>
<td>An illustration of direct derivation</td>
<td>74</td>
</tr>
<tr>
<td>FIGURE 6.6</td>
<td>PARALLEL INDEPENDENCE OF DIRECT DERIVATIONS</td>
<td>75</td>
</tr>
<tr>
<td>FIGURE 6.7</td>
<td>SEQUENTIAL INDEPENDENCE OF DIRECT DERIVATIONS</td>
<td>75</td>
</tr>
<tr>
<td>FIGURE 6.8</td>
<td>ILLUSTRATION OF PARALLEL PRODUCTION</td>
<td>76</td>
</tr>
<tr>
<td>FIGURE 6.9</td>
<td>THE DIRECT DERIVATION IN THE SPO APPROACH</td>
<td>77</td>
</tr>
<tr>
<td>FIGURE 6.10</td>
<td>An algorithm for checking the dangling edge condition</td>
<td>78</td>
</tr>
<tr>
<td>FIGURE 6.11</td>
<td>A HOST GRAPH DEPENDENT DANGLING EDGE CONDITION</td>
<td>78</td>
</tr>
<tr>
<td>FIGURE 6.12</td>
<td>A SUBGRAPH FOR PROPOSITION 6.1</td>
<td>79</td>
</tr>
</tbody>
</table>
FIGURE 6.13 THE DOUBLE PUSHOUT OF THE DIRECT DERIVATION ........................................................... 80
FIGURE 6.14 THE DPO EXTENDED WITH THE METAMODEL ELEMENTS................................................. 80
FIGURE 6.15 CATEGORY DIAGRAM WITH PARTIAL MORPHISMS AS INCLUSIONS ................................. 81
FIGURE 6.16 THE DIAGRAM OF THE TRANSFORMATIONS STEP WITH METAMODELS .............................. 81
FIGURE 6.17 A DOOLITTLE DIAGRAM WITH INCLUSIONS IN SET ......................................................... 81
FIGURE 6.18 SEPARATED PUSHOUT FOR METAMODEL-PRESERVING DIRECT DERIVATIONS ............ 83
FIGURE 8.1 THE OVERVIEW OF THE TRANSFORMATION PROCESS ......................................................... 87
FIGURE 8.2 STATECHART METAMODEL IN VMTS .................................................................................. 88
FIGURE 8.3 A SIMPLE STATECHART EXAMPLE ..................................................................................... 88
FIGURE 8.4 ATTRIBUTE INSTANTIATION IN VMTS FOR METASTATE-STATE .......................................... 89
FIGURE 8.5 THE CodeDOM METAMODEL ............................................................................................... 90
FIGURE 8.6 THE TRANSFORMATION RULE FOR THE STATECHART MODEL TRANSFORMATION ............ 91
FIGURE 8.7 XSL SNIPPET CREATING A NEW METHOD FOR A STATE .................................................... 92
FIGURE 8.8 THE RESULT OF THE TRANSFORMATION PROCESS ............................................................... 92
FIGURE 8.9 TRANSFORMATION RULE: STEP 2 ....................................................................................... 93
FIGURE 8.10 XSL FOR ATTRIBUTE TRANSFORMATION ............................................................................ 94
FIGURE 8.11 FEATURE DIAGRAM META-MODEL ..................................................................................... 95
FIGURE 8.12 NORMALIZATION STEP 1: FEATURE DIAGRAM WITH ONE OPTIONAL ALTERNATIVE FEATURE IS NORMALIZED INTO A DIAGRAM WITH ANY OPTIONAL ALTERNATIVE FEATURES ......................................................... 96
FIGURE 8.13 NORMALIZATION STEP 2: FEATURE DIAGRAM WITH ONE OPTIONAL OR-FEATURE IS NORMALIZED INTO A DIAGRAM WITH ANY OPTIONAL FEATURES ......................................................... 96
FIGURE 8.14 A FEATURE DIAGRAM FOR MOBILE PHONE SOFTWARE ................................................... 97
FIGURE 8.15 THE RESULT OF THE NORMALIZATION PROCESS ............................................................... 97
FIGURE 8.16 SEQUENCING THE NORMALIZATION PROCESS ................................................................. 98
Abstract

Software model transformation is a key issue in modern software engineering. Currently, new approaches, methods and techniques are being researched built intensively on expressive and efficient transformation systems.

There is a demand for researching the theoretical and practical foundations of these transformation systems. This work addresses the unified treatment of metamodels and graph rewriting-based transformation systems, in order to (i) provide a common language for both application fields, (ii) provide a rich visual language for model transformation and (iii) benefit from the additional information stemming from the availability of metamodels. The related results that underpin the realization of such a system can be divided into three main parts.

The first theoretical contribution provides the algorithmic background to match metamodel-based transformation rules. Formulae are given to compute the number of objects in a valid instantiation of a UML class diagram from the multiplicity values for a single association. A worst case limit is proven for the number of objects to be examined to decide the valid instantiation. Then, the Incidence Matrix with Multiplicity (IMM) algorithm is developed to generalize the instantiation decidability for not only a single association but an arbitrary model. The correctness of this algorithm is also proven. Using the IMM algorithm, the VF2 algorithm is extended to the metamodel-based case, and we examine with optimization steps.

The second result group deals with mappings between the model and its metamodel in both directions. Conditions are provided to determine the existence of a homomorphic *typeof* mapping. An algorithm is given to transform a general metamodel to an equivalent metamodel, where the *typeof* mapping is homomorphism. Results are given to decide the applicability of a transformation rule offline. Then topological relationships are proven between compatible and partially compatible metamodels. As far as the *instanceof* relationship is concerned, a decomposition algorithm is developed for the instance graph, where the instantiation relationship between the decomposed submodel and the metamodel is homomorphism. The complexity of the algorithm is also calculated. The results built on the homomorphic mappings can be used for offline analysis of transformation steps.

The third contribution includes the categorical examination of the rewriting rules. Generalizing the Double Pushout (DPO) approach to the metamodel-based case, categorical equations are proven for the most important transformation cases. Using the result of the DPO approach, the sequential and parallel independence is generalized to metamodel-based rules. With the help of these propositions the parallel execution and sequential ordering can be analyzed by tools.

In order to illustrate the practical applications of the results, a tool called Visual Modeling and Transformation System (VMTS) has been developed. Its application includes MDA model compilers for UML class and statechart diagrams as well as normalizing feature models for the Generative Programming (GP) paradigm.
ÖSSZEFOGLALÓ

A szoftvermodell transzformációja kulcsfontosságú szerepet tölt be a modern szoftverfejlesztés területén. Napjainkban új megközelítések, módszerek és technikák születnek, amelyek építenek a jó kifejezőképességű, hatékony transzformációs rendszerek tulajdonságaira. Ezért szükség van ezen rendszerek elméleti és gyakorlati hátterének kutatására. Jelen munka kidolgozza a metamodellek és a gráf újraírás alapú transzformációs rendszerek egységes kezelését, abból a célból, hogy (i) egy közös nyelvet alakítson ki mindkét alkalmazási terület számára, (ii) egy gazdag vizuális nyelvet biztosítsan a transzformáció számára, és (iii) kihasználja azt a többletinformációt, amely a metamodellek rendelkezésre állásából származik. Azok az eredmények, amelyek lehetővé teszik egy ilyen transzformációs rendszer megvalósítását, három részre oszthatók.

Az első elméleti eredménycsoport algoritmikus hátteret biztosít a metamodell alapú transzformációs szabályok baloldalával izomorf részgráf megkeresésére. A multiplicitásértékek alapján összefüggéseket bizonyítunk be az UML osztálydiagram érvényes példányosításában előforduló objektumok számának meghatározására egy asszociáció esetén. Megadunk egy, a legrosszabb esetre érvényes határértéket a megvizsgálandó objektumok számára, amely az érvényes példányosítás előírásainak szükségességét az UML osztálydiagram észlelésére a megvizsgált példányosításokra mondunk  be. Ezek után kifejlesztjük az Incidence Matrix with Multiplicity (Illeszkedési Mátrix Multiplicitással, IMM) algoritmust, amely általánosítja példányosítás elődöntését egyetlen asszociációóról, illetve tetszőleges modell gráfról. Az algoritmus helyessége szintén bizonyítást nyer. Az IMM algoritmus felhasználásával kiterjesztjük a VF2 gráf izomorfia eldöntő algoritmust a metamodell alapú esetre az optimalizálási lépések vizsgálata mellett.

Az eredmények második csoportja a modell és a hozzátartozó metamodell között létesíthető leképezésekkel foglalkozik mindkét irányt figyelembe véve. Feltételeket adunk meg annak eldöntésére, hogy létezik-e homomorf leképezés, és egy algoritmust fejlesztünk ki, amely áttranszformál egy általános metamodellt egy vele ekvivalens metamodellt, ahol a typeof leképezés homomorfizmus. Ezek után topológiai kapcsolatokat állapítunk meg kompatibilis, illetve részlegesen kompatibilis metamodellek között. Az instanceof leképezésre egy dekompozíciós algoritmust adunk meg, ahol a példányosítás leképezése a részmodellek és a metamodell között homomorfizmus, Mindemellett meghatározzuk az algoritmus komplexitását is. A homomorf leképezés alkalmazásával nyert eredmények a transzformációs lépések transzformációs időn kívüli offline elemzésére használhatók.

Az eredmények harmadik csoportja a metamodell alapú transzformációk alapján összefüggéseket bizonyít a metamodell alapú esetre. Feltételeket adunk meg annak eldöntésére, hogy létezik-e homomorf leképezés, és egy algoritmust fejlesztünk ki, amely áttranszformál egy általános metamodellt egy vele ekvivalens metamodellt, ahol a typeof leképezés homomorfizmus. Ezek után topológiai kapcsolatokat állapítunk meg kompatibilis, illetve részlegesen kompatibilis metamodellek között. Az instanceof leképezésre egy dekompozíciós algoritmust adunk meg, ahol a példányosítás leképezése a részmodellek és a metamodell között homomorfizmus. Mindemellett meghatározzuk az algoritmus komplexitását is. A homomorf leképezés alkalmazásával nyert eredmények a transzformációs lépések transzformációs időn kívüli offline elemzésére használhatók.

A harmadik eredmény az újraírású szabályok kategóriaelméleti vizsgálata tartalmazza. A Double Pushout (DPO) megközelítés eredményeit általánosítva a metamodell alapú esetre feltételeket adunk a szabályok alkalmazhatóságának transzformációs időn kívüli offline eldöntésére. Kategóriaelméleti egyenleteket állítunk fel a transzformáció fontosabb esetére. A DPO megközelítés eredményeit felhasználva általánosítjuk a soros és a párhuzamos futtatási és átváltási eszközök képesek a párhuzamos futtatás és a soros felcserélhetőség lehetőségét.

Preface

Dedication
The content of this thesis is a product of the author’s original work except where explicitly stated otherwise.

Nyilatkozat
Alulírott Levendovszky Tihamér kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem. A bírálatok és a védésről készült jegyzőkönyv a későbbiekben a Dékáni Hivatalban elérhetőek.

Budapest, 2005. augusztus

(Levendovszky Tihamér)
Acknowledgements

This thesis could not have been created without the support of many people. First of all I am really grateful to my family, who has been standing by me for all my life. Thanks to my friends, the time spent together helped me to start this work again and again. I am indebted to my advisor, Hassan Charaf, for facilitating the human and the financial conditions of the work; to István Vajk for sharing the experiences enthusiastically, and to both of them for stressing a practical engineering attitude towards the problems. I thank my uncle, János Levendovszky, for the stimulating discussions, his support and for showing the directions constantly towards the “strategic goals”.

I am indebted to the members of our research group, László Lengyel and Gergely Mezei, who have implemented and many times inspired the theoretical results. I would like to thank the Applied Computer Science Group at the Department of Automation and Applied Informatics for creating the dynamic atmosphere of “fighting together”, which has promoted my research much, and also Renáta Iváncsy, Jenő Hetthéssy and Wade Whisenant for proofreading this work. I am grateful to my students, especially László Angyal, for their work.

This work could not have happened without the time I spent at the Institute for Software Integrated Systems, at Vanderbilt University, Nashville, Tennessee. Therefore, I would like to thank to my supervisor there, Gábor Karsai, who helped me in the first steps of my research. Moreover, I am grateful to Ákos Lédeczi and Péter Völgyesi, who introduced me to GME, and to other colleagues, Endre Magyari, Aditya Agrawal, Feng Shi, Jonathan Sprinkle and Miklós Maróti for the stimulating discussions on Friday afternoons.

The second version of this work was completed in Berlin, where I was a SEGRAVIS grant holder. I am very grateful for this opportunity, and the help of the people working there and the other grant holders, especially Hartmut Ehrig, Gabi Taenzer, Olga Runge, Dániel Varró, Juan de Lara.

I am indebted to my high school teachers, especially to Károly Kőváry and Alfréd Simonyi, who have instilled in me a view and way of thinking still valid and useful. And last, but not least, I would like to thank to my coauthors.
1. Introduction

The application of visual languages (VL) plays a key role in modern software engineering. A wide variety of models is used across the life cycle of a software system to enforce faster and more efficient requirements analysis, design and testing. Compared to textual languages, VLs are no less formal [Harel & Rumpe 2000], and in practice they have been proven to be more comprehensible and convenient under many circumstances.

One of the most important aspects of the VLs is the way in which they are processed. After creating models, there is a natural demand for processing them automatically. In most cases, this means producing code in a specific programming language. This leads to the need for software model transformation, which is a general means to extract the information from the model in order to reformulate it in an appropriate representation. Currently, new approaches, methods and techniques are being worked out, built intensively on expressive and efficient transformation systems (c.f. Chapter 3). To make these achievements reachable for industrial applications, new transformation principles need to be researched and tested via case studies and in industrial software development processes. Compared to other disciplines, such as information theory, this field is relatively new, lacking a general and unquestionably distilled formal background. The contributions included in this thesis are intended to promote this theoretical side at the same time, always maintaining strong connections with the practical applications.

1.1 Thesis Contributions

Since this work is intended to be a thesis pertaining to computer science, not from mathematics, efforts have been made to turn the sometimes abstract mathematical terms and lines of thought into a digestible reading for engineers.

- The unusual mathematical definitions are briefly summarized to make this work a self-contained reading for engineers with different backgrounds.
- The application and the motivation are always presented, and they always precede the formal models and the theoretical proofs and results.
- The results are approached via a series of intuitive thoughts, sometimes at a price of giving two proofs for the same proposition.

This thesis is organized around two applications of visual modeling languages.

- Using visual languages to specify a highly configurable visual language environment for system modeling and design.
- Using visual languages to specify the transformation of visual languages.

Illustrating the achievements, a software system called Visual Modeling and Transformation System (VMTS) has been developed.

In order to create a real-world transformation system an algorithmic background needs to be worked out. Based on this sort of results a transformation system can be built which is able to execute the transformation steps described by the formal model. Here the pattern matching or subgraph isomorphism is dealt with primarily considering the necessary extensions for metamodel-based transformation rules.
A formula is provided, which reveals the dependency between the multiplicities of the associations and the allowed number of objects in the object diagram.

- A worst case algorithm is given to decide the capability of instantiation.
- An algorithm and a representation are provided to generalize the dependency between the association multiplicity and the allowed number of objects to an arbitrary class diagram.
- The correctness of the algorithm is also proven.
- A matching algorithm is developed.

The homomorphic *typeof* relationship helps one understand the instantiation process better; a closer connection can be established between the model and the metamodel. Topological validation of the metamodel-based rules can be performed, built on the following results:

- Conditions are proven that serve as criteria for the existence of the homomorphic *typeof* mapping.
- An algorithm is provided to create a homomorphic instantiation for an arbitrary metamodel. It is proven that the output of the algorithm is equivalent to the input metamodel, and there exists a homomorphic *typeof* relationship between the mode and the metamodel.
- Topological properties are proven for the created homomorphic metamodel.
- The reverse direction is also discovered.

The formal specification of the transformation is a crucial part of this work. The theoretical results of this part are twofold. Firstly, we create a mathematical background which facilitates further analysis and the development of additional theorems. Secondly, a transformation method has been worked out which applies fully UML compliant concept of metamodel, so it is easily conceivable and applicable by practitioners already familiar with UML-like modeling. Propositions proven in this part allow validation methods and analysis of concurrent execution.

- It has been proven that the DPO approach can be generalized to a class of homomorphic metamodels.
- When the metamodel remains constant during the transformation, another categorical equation is proven.
- For these metamodels, the DPO Parallelism Theorem is generalized.

The practical relevance of the first problem group is an algorithm for metamodel-based transformation rules. However, some of the results can be applied through general UML tools checking the conformance between the metamodel and the model.

As illustrated in this thesis, results related to the second problem group help to use special properties, which can be derived from an arbitrary metamodel and applicable to topological validation.

The application of the third group is the parallel execution for metamodel-based rewriting rules.
1.2 Thesis Structure

The rest of this thesis is organized as follows.

- Chapter 2 is devoted to illustrate the motivations of the results in this thesis. This chapter serves as a general introduction to the topic of metamodeling and model transformation. It also outlines the most important approaches to model transformation.

- A model transformation system is introduced in Chapter 3, together with the related work. Notions of metamodeling tools, model transformation systems, and Model-Integrated Computing are shown. The purpose of this chapter is twofold: (i) it illustrates the concepts of a metamodel-based transformation system and (ii) it raises open issues to implement such a system. The focused tool is called Visual Modeling and Transformation System (VMTS), and it has been developed at the Department of Automation and Applied Informatics of Budapest University of Technology and Economics.

- In Chapters 4-6 we present the contributions of this thesis. These chapters are structured in the following way: firstly, motivations are presented, secondly, the related work is elaborated, and finally, the contributions are discussed. The results are divided into three parts: (i) the matching algorithm and related propositions (Chapter 4), (ii) mappings between the model and metamodel elements and (Chapter 5) (iii) metamodel-level categorical constructs are introduced along with the corresponding propositions related to parallel rule execution (Chapter 6).

- In Chapter 7 the evaluation of the theoretical results can be found.

- To illustrate the practical relevance, the application of the results is shown in Chapter 8. Out of many possible applications three have been selected: (i) code generation from statechart diagrams, (ii) code generation from class diagrams and (iii) normalizing feature models. The first two examples are taken from the popular field of Model-Driven Architecture (MDA) and the third one is a contribution to Generative Programming (GP).

- The last chapter (Chapter 9) is devoted to the summary and the outline of future work.

- In the Appendix some of the detailed proofs are given along with a comparison between the mainstream metamodeling and model transformation tools in table format.
2. Motivation

Graph-like structures has been proven to be useful for modeling in several fields. Graphs have been used not only to visualize electric circuits, but to deliver the computational models of these networks. For instance, the MathWorks’ Simulink Toolbox for MATLAB [MathWorks] is a widely used software package both in the field of control theory and in developing real-time systems for the automotive industry. In a way similar to the applications above, software technology strongly builds on graph-based models.

The further part of this chapter is devoted to review the industrial demands and backgrounds which have led to the extensive research of software model transformation methods, and the need for metamodel-based transformation systems. An overall view is provided here, as opposed to a narrow concentration on the closely related topics in the following chapters. First, we briefly introduce the concept of metamodeling and its significance. Then some applications are described where model transformation is a crucial and inevitable part of the specific field or technique discussed. Finally, model transformation techniques are elaborated, and the open issues are concluded before the chapter summary.

2.1 Models and Metamodels

The software industry has applied graph-like structures to a large extent from almost the beginning of the field. A wide variety of flowcharts and data flow diagrams were the first structures to describe the tasks implemented by the system under design. The structured software development paradigm has adopted these structures. In addition, several other diagrams have been added (e.g. Jackson diagrams [Jackson 1975] [Jackson 1983]). The proliferation of the object-oriented paradigm resulted in the evolution of many different and sometimes contradictory models and notation systems e.g. [Rumbaugh et al.1990] [Jacobson et al. 1992] [Booch 1993]. This situation was solved by the introduction of the standard Unified Modeling Language (UML) [OMG UML] in 1998.

2.1.1 The Unified Modeling Language

“The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system” [OMG UML 1.5]. Version 1.5 (released in 2003) of the UML standard contains nine diagrams listed below:

- Use Case Diagram
- Static Structure Diagrams
  - Class Diagram
  - Object Diagram
- Behavioral Diagrams
  - State Diagram
  - Activity Diagram
- Interaction Diagram
  - Collaboration Diagram
  - Sequence Diagram
- Physical Diagrams
  - Component Diagram
  - Deployment Diagram
The use case diagram is a semiformal description of the functional requirements. Static structure diagrams depict classes, objects and their relationships. Behavioral diagrams model the dynamic behavior of the classes and objects. Interaction diagrams concern the communication scenarios between objects. Physical diagrams express the arrangement of the source files and the deployed system components. These diagrams and concepts are not delineated here. For a comprehensive approach please refer to [Fowler & Scott 1999], an overall reference can be found e.g. in [Rumbaugh et al. 1999]. Moreover, action semantics concepts borrowed from SDL [ITU SDL] have been added to the UML 1.5 standard [OMG UML 1.5], which can be considered as a new diagram type, where the visual appearance is left to the modeling tool, and only the semantics is specified.

UML diagram structure suggests that software models are composed of often orthogonal diagrams with fundamentally different concepts describing distinct aspects of a software system.\footnote{In addition to the standard UML diagrams Kruchten defines the 4+1 view of a software system architecture [Kruchten 1995], which claims to be orthogonal, and some of this view is in one-to-one correspondence with a specific UML diagram.} Moreover, UML has very efficient extension mechanisms built-in, like constraints and stereotypes. These mechanisms can be used to construct profiles for business modeling or any arbitrary domain.

Even the most popular modeling language changes a lot; thus several parts must be rewritten if it is hard wired in the modeling environments. UML 2.0 [OMG UML 2.0] [Fowler 2003] [Pender & Pender 2003] [Eriksson et al. 2004] has renamed the collaboration diagram to communication diagram, and added four more diagrams: composite structure diagram, interaction overview diagram, timing diagram and package diagram, totaling thirteen diagrams. In addition, there are a growing number of standard (e.g. [OMG EDOC] [OMG SPT] [OMG EAI] [OMG TP] [OMG CP]) and non-standard (e.g. [Aldawud et al. 2001]) profiles for UML diagrams, which need to be adopted by modeling tools in a flexible way.

Beyond UML there are fields where special languages are needed, since UML would be too general for them. For this reason, Domain Specific Languages (DSLs) [Czarnecki & Eisenecker 2000] continue to play an important role in modeling software/hardware systems. Therefore the more configurable a modeling tool is, the more likely it is to adopt the changes of the standard modeling language and to be capable of expressing DSLs.

The following questions arise as a result of the aforementioned issues regarding the evolution and nature of UML:

- Is there a common basis for these distinct modeling concepts? If so, can we use it to describe all the UML models?
- Is it possible to build a flexible visual tool supporting all the existing models and facilitating the creation of arbitrary DSLs?
- How can we maximize the reusability of the already existing models?

The next sections are designed to provide answers to these questions.

### 2.1.2 Instantiation

The concept of metamodeling is founded in the roots of the object orientation. Based on their applications, as in [Sztipanovits & Karsai 1997] [Atkinson & Kühne 2001] [Atkinson & Kühne 2002] [Atkinson & Kühne 2003], we can aptly consider it a really fortunate construct. It serves as a descriptive and type-safe basis for a large
class of models. For metamodeling tools it simplifies the creation of generic environments modeling either UML or other DSL.

In tracing the evolution of programming languages, the concept of metamodeling is quite natural. With the emergence of the object-oriented languages, it is mainly the programmer who creates and designs the type system and the properties of the individual types used by the program, rather than exclusively accommodating the predefined types offered by a specific programming language. Hence, the program design also involves the design of the type system. The individual type is called class in an object-oriented language. Please refer to e.g. [Albert et al. 2004] for an implementation-based approach of this concept. When a class is instantiated at runtime, the created instance is referred to as an object. An object can hold a reference to another object, and this relationship is called link.

The corresponding construct on the metamodel level is an association which connects classes and specifies the possible number of its instantiating links. Figure 2.1 depicts a class diagram and its schematic instance.

![Figure 2.1 Metamodel (a) and its corresponding model (b)](image)

With the proliferation of UML, this concept has also been standardized inherently. In summary, metamodels are related to the instance model as the UML Class Diagram is related to the UML Object Diagram, though the Object Diagram is visualized in an arbitrary (and also domain specific) way.

Later in Chapter 3 of this thesis we touch upon metamodeling on a more formal basis, but only an intuitive introduction is given here. Our example is simple: we would like to create a metamodel for UML State Diagrams. A composite picture is depicted in Figure 2.2, where an arrow denotes the instantiation. The corresponding views are illustrated in Figure 2.3 and Figure 2.4, respectively.
As it can be seen in Figure 2.4, a state diagram consists of states (including two special ones: the start and end state) and transitions. The corresponding metamodel is depicted in Figure 2.3.

The metamodel specifies an abstract TransitionEndPoint, which is the base for all elements that can be connected by a directed transition (SynchronizationBar, State). A State can contain zero or more TransitionEndPoints.
On the topmost layer (Figure 2.4) a simple statechart is depicted, which models a door opening system asking for a two-digit code. There is a 5 second time limit for entering the code, and in case of an erroneous input, the device can be cleared to start typing again. On successful entry of the code, the protected door is opened.

### 2.1.3 Metamodeling and UML

The primary document specifying the standard for metamodeling is OMG’s Meta Object Facility (MOF) [OMG MOF 2002] [OMG MOF 2.0]. This also serves as the foundation for the UML metamodel. The structure provided by MOF can be depicted as a pyramid (Figure 2.5).

We use the UML object diagram to illustrate the individual layers. The information layer (M0) is the UML object diagram layer. An element can be an UML object (this is usually created by the operator `new` in the most frequently used OO programming languages). The UML class diagram resides on the model layer (M1); an example element can be a class named `MyApplication`. The metamodel layer (M2) contains, for instance, metaclasses. A hard-wired meta-metamodel (M3) follows, which defines the basic visual vocabulary for the framework. The four-layered architecture is a skeleton only, not a recommendation. The number of the layers may vary. Our previous statechart model (c.f. Section 2.1.2) uses only a visual vocabulary and two other layers. MOF specifies at least two layers, but there is no upper limit for the number of layers.
Unfortunately, MOF raises unresolved problems related to the instantiation relationship. As it has been pointed out in [Kurtev & van den Berg 2004], MOF implicitly uses several instantiation relationships: relationships between levels M3, M2 and M1 apply a different instantiation relationship compared to those between M1 and M0. Moreover, in [Atkinson & Kühne 2001] there is a concept called deep instantiation as opposed to shallow instantiation. The “deepness” here means that a metamodel element can affect not only its immediate instance layer, but also - based on a potency value - other layers in the instantiation chain. A solution is introduced in [Atkinson & Kühne 2002] [Atkinson & Kühne 2003] to separate ontological metamodels and linguistic $instanceOf$ relationships. These concepts are not handled within the MOF framework.

The structure of the UML 2.0 metamodel has changed, in several aspects, since its preceding standard. The two main motivations were the appropriateness for Model-Driven Development (discussed in Section 2.2.1) and the demand for preciseness. UML 2.0 is powerful enough to specify a full-fledged software system; as an example, there are serious changes made in the activity diagram by introducing ports and tokens. As far as the precision is concerned, the UML metamodel itself has become more precise and better-structured. The document containing foundations of the language constructs is called the UML Infrastructure [OMG UML 2.0a]. It defines the visual language used by MOF 2.0, which is under finalization, and is used by the UML Superstructure [OMG UML 2.0b], which defines the actual UML modeling languages on the basis of the infrastructural elements. Based on the general constructs defined by the UML Infrastructure can also be used in metamodeling Domain Specific Languages. In fact, the UML Infrastructure contains the metamodeling foundation of UML, which was a part of the MOF standard in the previous version of the standard. Unfortunately, the UML Infrastructure inherited the shortcomings discussed above, with respect to MOF instantiations, and changes the instantiation relationship between different layers. The standard does not cite any tools that support all the language concepts described in the infrastructure; thus, one can expect the usual gap between the practical metamodeling tools and the metamodel-based foundation of the standard modeling language.

This conclusion is also drawn by the Precise UML Group [PUML], whose mission is to:

- clarify and make precise the semantics of UML.
- reason with properties of UML models.
- verify the correctness of UML designs.
- construct tools to support the rigorous application of UML.

According to [Clark et al. 2000] a metamodel-facility consists of two parts:

- A Metamodeling Language (MML)
- A tool supporting MML

Based on MML, the question of instantiation must be unambiguously determined having a model and a metamodel. In other words MML should describe the syntax, semantics, and the mapping between the languages. The syntax is the shape of the elements of the visual language: “how they are rendered”. The semantics is defined by the instantiation: whether a model is a correct instance of a metamodel, and where a mapping defines the instantiation relationship. MML includes:

- Constructs for specifying object structures (e.g. class diagrams)
- Constructs for expressing well-formedness constraints on object structures (e.g. OCL).
- Constructs for packaging and composing fragments of language definition (e.g. packages and their relationships).
- Constructs to support reflection.

The tool should provide facility to check the instantiation or generate an instance of a metamodel. For the syntax, a graphical configuration is needed to render the shapes correctly. The MMT (metamodeling tool) prototype can be accessed at [MMT].

Having discussed the application and the problems concerning metamodels, the demand and the need for model transformation is introduced.

2.2 The Claim for Software Model Transformation

The software model transformation has its roots in the early periods of computer science applied to compilers, Petri Nets, flow charts, among others. In this section we overview some of the fields within modern software development which explicitly require certain model transformation techniques.

![Figure 2.6 The most prevalent paths of software model transformations](image)

Figure 2.6 depicts the most prevalent applications of model transformations in the field of computer science. Although these are the most common transformation paths, there are several other possibilities depending on the specific application. A path leads from the models to the implementation: this illustrates the efforts towards code generation. It may involve other graph-like representations, as with abstract syntax trees (AST) [Aho et al. 1986], or code representation trees, like CodeDOM [Dollard 2004] [pp. 781-792 in Albert et al. 2004] and the Java Document Model of the Eclipse JDT Core [Budinsky et al. 2003]. Graph transformation techniques are also used in formal model checking. This section summarizes the MDA-related issues, the claims for software model transformation from the code generation point of view as well as from the aspect of formal model testing.

2.2.1 Model-Driven Architecture

Research projects similar to that reported in [Harel & Gery 1997] managed to raise the execution environment to the modeling level. Model-Driven Architecture [OMG MDA] [Mellor & Balcer 2002] [Mellor et al. 2004] [OMG MDA Guide] [Frankel 2003] [Kleppe et al. 2003] allows software developers to create a system entirely comprised of full-fledged models. This concept is also referred to as Model-Driven Development (MDD). These models can also include textual languages; they consist of several well structured, maintainable parts. The MDA way for software development is essentially transforming an input model provided by the developer to one or more output models which are usually closer to the source code representation of the system, than to the input model. Illustrating these concepts the input model is
called **Platform Independent Model** (PIM), and the output model is referred to as **Platform Specific Model** (PSM).

Consequently, model transformation is unquestionably the most crucial part of MDA. MDA Guide [OMG MDA Guide] defines model transformation as follows:

"Model transformation is the process of converting one model to another model of the same system."

The definition above implies that both PIM and PSM must model the same system. Therefore, it raises an interesting question as to whether MDA transformations should preserve the meaning of the input model, i.e. whether the transformation should preserve semantics. According to [Kleppe & Warmer 2003] this equivalence of the two models cannot be treated easily for several reasons:

- UML itself, which is the most widely used specification language for PIM, does not have formal semantics yet (semantics is described in English). Therefore, it is unreasonable to ask whether a language with no formal semantics (i.e. with no meaning for automatic checking) can be transformed to another language having the same semantics (meaning).
- In practice, it is quite hard to compare two languages with different formal semantics, typically semantics of a modeling language (UML with OCL [OMG OCL] [Warmer & Kleppe 2003]) and the operational semantics of a programming language (e.g. Java).
- In practice, two systems can be considered identical, formally cannot. For example, modifying the access modifiers in C# from private to public makes getter and setter functions superfluous; the interface has also changed, but essentially we have the same meaning.

Furthermore, if the source language of a transformation has no formal semantics, but the target language does, the transformation defines the semantics (in terms of the target language). If both the source and the target language have semantics then the transformation defines a mapping between the two semantics.

At least for the time being, the formal semantics for UML-based models can be considered an unresolved open problem; they are beyond the scope of this thesis. As a result, we leave the transformation semantics to the developers. Our objective is to create an expressive and general transformation method and the related software system, which is based on the practical usability rather than formal semantics.

### 2.2.2 Transformations in CASE Tools

Naturally, CASE tools are the most widely applied area of model transformation, thus the related aspects cannot be summarized in such a work like this thesis. Even so, this introductory subsection, however, presents some examples to illustrate the need for the model transformation systems which serve code generation purposes. The Object Modeling Technique (OMT) [Rumbaugh et al.1990] methodology has a branch for database design (OMT DB) [Blaha & Premerlani 1996] [Blaha & Premerlani 1998]. The creators of the OMT DB methodology have described a few transformations in English to clarify the object model (similar to UML class diagrams). They clearly state the need for a formal mechanism to perform these model manipulations automatically. The MATLAB’s control system toolbox [MathWorks] allows specifying control systems visually; then, it is able to run a simulation for the
modeled system. Several programming environments offer visual editing of user interface elements, generating code from these models. The Ptolemy II [Ptolemy] is a set of Java packages designed for modeling heterogeneous, concurrent systems. The interactions between the entities are called models of computation, the central concept of the tool. Further examples of the model of computation are discrete-event systems, dataflow reactive systems.

### 2.2.3 Transformations and Model Testing

This section is devoted to the methods for verifying functional correctness of a system. As summarized in [Slonneger & Kurtz 1994], the semantic description of a programming language can be achieved with the following techniques:

- Translational Semantics
- Operational Semantics
- Denotational Semantics
- Axiomatic Semantics
- Algebraic Semantics
- Action Semantics

The *translational semantics* describes a language by translating it into another, well-defined language. The operational semantics approach defines the operation (the steps performed by the program) through operations of an abstract machine (e.g. Vienna Definition Language, VDL or Stack, Environment, Control, Dump, SECD machine). Abstract state machines [Börger & Staerk 2003] have received much attentions recently. The formal background is given by evolving algebras created by Gurevich [Gurevich 1988]. This approach is a translational semantics technique due to the notion of refinement. Refinement is, in a sense, the opposite of abstraction. Assuming that an ASM \( A \) is refined to a machine \( B \), a partial abstract function \( F \) is a proof map which maps certain refined states \( \beta \) of \( B \) to abstract states \( F(\beta) \) of \( A \). Certain sequences \( R \) of \( B \) rules are mapped to sequences \( F(R) \) of abstract \( A \) rules, such that the following categorical diagram commutes [Börger 1999] (Figure 2.7).

![Figure 2.7 ASM refinement](image)

A detailed explanation of commuting categorical diagrams can be found in Section 6.1. Through the concept of refinement, one can think of refined abstract runs as a special case of the abstract runs. Therefore, if the abstract runs are proven to be correct, the refined rules are also correct with respect to the properties to be proven.

The *structural operational* semantics uses deductive systems instead of abstract machines.

*Denotational semantics* maps program constructs to mathematical objects. According to [Harel & Rumpe 2000] the syntax of a visual language is the representation, the notation itself, while the semantics is defined by a semantic domain and a mapping between the syntactic constructions and the semantic domain. This mapping is referred to as semantic mapping. The elements of the semantic domain provide the “meaning”, and the semantic mapping assigns the “meaning” to
the elements of the notation. For instance in an algebraic expression like $x+y$, we define

- the syntax with a BNF (Backus-Naur Form) grammar
- the semantic domain can be the set of the real numbers
- the “+” operator can be assigned to the mathematical addition

The treatment of the visual languages are very similar to that of the textual languages, but it needs to be taken into consideration that the syntactic constructs are diagrammatic and/or iconic in nature, which makes it less trivial to find and define the appropriate semantic mapping. For VLs, graph transformation methods can provide a formal tool to specify this semantic mapping.

**Axiomatic semantics** is based on methods of logical deduction from predicate logic and maintains assertions about invariant relationships, which must be true during all executions of the program. These invariant assertions are realized as pre- and postconditions. Preconditions must hold before an operation, postconditions must be true after an operation. This approach does not depend on abstract machine-like constructions as the previous methods. If the initial and the final assertions of two programs are the same, they are considered equal.

**Algebraic semantics** uses abstract data types and operations on them. The characteristic properties are described by algebraic axioms based on abstract algebraic constructs.

**Action semantics** (should not be confused with UML action semantics) is similar to denotational semantics, but it maps the language constructs onto first-order entities instead of mathematical objects. The first-order entities are actions (embodies the computational behavior), data (pieces of information, defined in an algebraic manner) and yielders (evaluates data based on the current information).

As it has been mentioned earlier in this section, graph transformation methods are widely applied to define formal semantics. Wagner and Gogolla defines operational semantics for object oriented languages (pp. 181-211 in [Ehrig et al. 1999a]) via a textual object description language called TROLL light. The VIATRA approach [Varró et al. 2002] [Varró & Pataricza 2003] [Varró 2003] introduces the Visual and Precise Metamodeling (VPM), which provides a common, metamodel-based basis for engineering models and graph-like mathematical objects (e.g. abstract state machines, Petri nets). It also separates the dynamic and static model constructs, specifying the static concepts with rather algebraic formalisms and the operational semantics with graph transformation constructs. Programmed graph replacement systems (e.g. PROGRES) specify denotational semantics using fixpoint theory (pp. 479-546 in [Ehrig et al. 1999a]) to formalize the control structures.

VMTS currently supports semantic constructs in a sense that syntactically facilitates describing and checking constraints, but the modeler is responsible for the semantic part, for the actual meaning. That means that VMTS incorporate no real automatic support for semantics.

### 2.3 Model Transformation Techniques

Models are primarily formalized as labeled directed graphs. Considering software applications, there are two important categories of transforming a labeled directed graph. The simplest and the most universal way is the traversal approach. The other uses graph rewriting as a transformation mechanism. This chapter introduces these approaches, and takes the preparatory steps toward another aspect of the motivation: applying these already invented mathematical results to create a formal background
for metamodel-based transformation methods, and to extend the existing results with respect to the presence of the metamodel as the transformation and the modeling formalism.

2.3.1 Model Transformation and MDA

As it has been mentioned earlier, model compilers constitute a crucial part of the MDA architecture. Since model transformation systems can serve as MDA model compilers, MDA provides a really important application of model transformation results. The Model-Driven Architecture goes further: it offers a standard interface to implement model transformation tools. The transformation related part of MDA is the Query, Views, Transformation for MOF 2.0 [OMG QVT] [Gardner at al. 2003].

According to that specification three types of operations are provided:

- Queries on models
- Views on metamodels
- Transformation on models.

In this terminology a query is an expression that is evaluated over a model. It cannot change the model, it is fully declarative. Queries can be constructed using UML Action Semantics as well as OCL [OMG OCL]. They can be regarded parallel to XPath in XML.

A view is a model being derived from another model (base model) via transformation. Any change in the view or in the base model affects the other. The metamodel of the view can be different from that of the base model. Views can be read-only. A query can be considered as a restricted view.

A transformation generates a target model from the source model. Transformations can lead to either independent or dependent models. The difference is that in the first case there is no relationship between the target and the source model after the transformation process. The units of the transformation are called rules.

These constructs are basically interfaces: they can serve as a basis for both the traversal and visual approaches.

2.3.2 The Traversal Approach

The popularity and importance of visual languages raises the question of how to process the models created by the professionals. Using the Model Integrated Computing (MIC) [Sztipanovits & Karsai 1997] terminology, the question can be reworked to ask how the models are interpreted.

A pure imperative – and probably the most obvious and universal - technique is to traverse the graph applying the facilities of one or more programming language, and produce the required output. This approach is really proven and widely used. It is applied in Intentional Programming (IP) [Aitken et al. 1998] [Simonyi 1999] to perform transformations to build the whole program tree from different types of segments. Compilers [Aho et al. 1986] [Muchnick 1997] also fall into this category: they traverse the abstract syntax tree in order to optimize or generate the desired output code.

Traversing the models is fast. Although the simplicity of the programming is tool-dependent, the idea is straight and widely applied. The code performing the transformation, however, is not so clear and hard-to-maintain. In case of complex model transformation problems, this approach is really hard to control. If raising the
abstraction level to a visual model level turned out to be useful in case of software, it might help in case of the transformation code as well.

2.3.3 The Visual Approach

The visual approaches to model processing apply graph rewriting (e.g. Montanari 1970) [Ehrig 1979] [Blostein et al. 1995] [Ehrig & Taenzer 1996] [Rozenberg 1997] [Ehrig et al. 1999a] [Baresi & Heckel 2002]) techniques. Basically a graph rewriting system is a set of rules that transforms a graph instance to another graph. If we consider string grammars, production rules are usually provided in the following form:

\[
\text{left hand side } \Rightarrow \text{ right hand side}
\]

This means that if we want to fire a production rule, we must replace the occurrence of the left hand side string with the right hand side. Similarly in graph rewriting there are also production rules consisting of a left-hand-side (LHS) graph and a right-hand-side (RHS) graph. The graph to which a rule is being applied is called host graph. Firing a rule means to find an occurrence of the LHS graph in the host graph and to replace it with the RHS graph. The found occurrence is called match or redex\(^1\). Graph rewriting is a hybrid technique using mainly declarative constructs, thus it allows the developer to concentrate on the problem definition rather than the methods that solve the problem.

Graph rewriting has also been applied in several fields apart from software model transformation. Various applications cover database applications [Rodrges & King 1997], AOP aspect weaving [Assmann & Ludwig 1999], compiler optimization [Assman 2000], molecular biology [Rosello & Valiente 2005] and chemistry [Rossello 2004].

Unfortunately, the use of this method is limited: practice has shown that when one wants to produce code in a programming language, this method needs to be accompanied with some minor traversing code. Moreover, because of the complexity, this method is worse at performance than the traversal approach.

There are, however, advantages as well.

- This technique incorporates the advantageous properties of visual languages.
- The units of the transformation are not edges and vertices. This facilitates the creation of more complex steps, which does not fragment the problem or the realization of the intentions and concerns of the rule semantics, and allows assigning complex properties to the rules like constraints.
- Because the complex steps and the human-readability, this approach is suitable to describe more complex transformations.
- The termination [Ehrig et al. 2005a] and the result of the transformation can be predicted in case of different possible rule application branches (confluence) based on theoretical results e.g. [Ehrig et al. 2005b], which is useful in case of complex model verification systems.
- Furthermore, it is possible to prove certain properties of the transformation, regarding the rules only. This is really hard, if not impossible, using general traversing techniques.

\(^1\) The redex stands for reducible expression. This expression goes back to graph grammars, where – similarly to string grammars – the rewriting rules try to reduce a given expression.
2.4 Open Issues

Open issues related to the transformation representation: the semiformal or informal models cannot be transformed by any methods, but they are very relevant in industrial applications (e.g. use cases). How can one formalize them so that they can be transformed? Which part of these models can be formalized or can be automated by computers? These are really difficult problems, which will probably remain unsolved for quite some time; hence, they require human interaction until the solution has been found.

Open issues related to the transformation methods are as follows.

1. The transformation method issue. The main research topic is to find new methods that exhibit:
   a. Usability: if a method is easy-to-use and useful in practice and case studies can prove that fact.
   b. Completeness: every model can be transformed to any model using this method.
2. Algorithmic issues. Most interpretation and transformation steps cannot be completed in polynomial time. To make it faster heuristics should be worked out. Because these include problem-specific assumptions, it is really difficult to make general statements about them. Frameworks, however, which allow algorithm plug-ins, can help this situation.
3. User support issues. These features give the user a feedback about the transformation rules he is working on. This includes offline check of the transformation rules as well as suggestions for correcting them.
4. Execution issues. Discovering the facility of making the execution path parallel.

The main approaches to these issues are summarized in Section 3.3, along with our proposed solutions (Section 3.4).

2.5 Chapter Summary

We have seen the importance of software model transformation. There is a real practical demand for efficient mechanisms: CASE tools, model checking applications, and the wide-spread MDA standard. We have illustrated the applicability of metamodel-based tools: which unify and treat models with very different concepts on a common basis. We have two transformation approaches: the traversal approach and a graph rewriting method. Besides these techniques, another claim has appeared: the transformation needs a formal mathematical background to reuse the existing theoretical results. Because of the lack of the formal semantics in UML, a transformation method should be as general as possible: it needs to overcome the gap between the languages, and be able to support user-defined semantics buried in the transformation specification.

By this point the goal has become clear: we want to create a general software model transformation method on a formal background, with compliance to the UML standard, and the capability of using metamodeling and graph rewriting techniques.

Combining metamodeling and transformation, the techniques listed above seem to provide a promising direction. Among others the expected benefits are the following:

- Generality: as it has been shown metamodels serve as a common basis for a large group of models by using general elements with type information
in a general language (UML class diagram metamodel). Thus with the help of the metamodel we can create a general transformation engine.

- Applicability in practical tools: the use of metamodels seems to be natural and close to the human thinking. This plays an important role when we want to create an easy-to-use and a powerful tool.
- Facility of optimization: we can assume that the metamodel is also available and we can use it during the transformation.
- Formality: the mathematical achievements can be used and extended as a formal background for the metamodel-based transformation methods, e.g. parallel rule execution.
- Using standard techniques: the notations are close to the UML standard. This makes the learning curve acceptable for the users.

The issues are addressed by introducing an expressive rule formulation principle, namely, the **metamodel-based rewriting rules** (Chapter 3).

In the next chapter a model transformation system is introduced which exhibit the properties discussed above. This system is used to illustrate the constructs appearing in this thesis and to show the practical relevance of the developed results. Presenting the features and their use in VMTS, we can raise further motivation for the theoretical results, and hopefully, the early introduction of a real system makes the comprehension of the more demanding theoretical chapters easier.
3. Metamodelling and Model Transformation Systems

Several works have already been mentioned which influenced or motivated this thesis. This chapter intends to summarize the metamodelling and model transformation tools established by other researchers providing the results that this work is directly built on. The metamodelling background is elaborated along with the related model transformation tools. Finally, the Visual Modeling and Transformation System is presented.

3.1 Metamodelling Tools

VMTS is strongly built on the research and experiments included in existing systems. Generic Modeling Environment (GME) [GME] [Lédeczi et al. 2001] [Lédeczi et al. 2004] is a highly configurable metamodelling tool supporting two layers: a metamodel layer and a modeling layer.

A part of the GME 4 metamodel is depicted in Figure 3.1.

![Figure 3.1 The GME metamodel (subset)](image)

An FCO (First-Class Object) is a common abstract base for Model and Atom elements. The difference between the Atom and Model elements is that Atom can be a leaf only in the containment hierarchy. Set is a general form of reference: it can refer to its element via the SetMembership connection. A Reference can be a visual alias for an FCO, which can substitute the referenced FCO. A Connection represents an edge between two FCOs, however a third one can be specified as an “association class”, and it must be contained by the model which can contain the Connection. The graph database part of GME is called Multigraph Architecture (MGA).

GME can be used for metamodel editing. When the metamodel is the edited model, a traversal processing procedure (MetaInterpreter) converts the edited model into a metamodel layer for the model editing usage. GME interpreters are analogous to VMTS TMPs. The model and the metamodel layer in GME cannot be treated uniformly, as in the two-layered Universal Data Model (UDM) [Magyari et al. 2003], where the metamodel and the model layer are implemented with the same data structure. This concept has been generalized in VMTS to an n-layer metamodelling environment.

UDM is a reflective environment implemented in C++ to support tool integration. Generated from a UML class diagram, a C++ API is available at run-time, which
METAMODELING AND MODEL TRANSFORMATION SYSTEMS

provides the type information for the object network. Along with these features UDM has a repository function, which allows storing and loading the objects created at runtime. Another unique contribution of UDM is that the metamodel and the model level can be stored exactly the same way in the internal data structures and with respect to the exposed interfaces.

The Eclipse Modeling Framework (EMF) [Budinsky et al. 2003] is a modeling environment for the Eclipse toolset. The simplified version of the EMF metamodel is depicted in Figure 3.2.

The Eclipse Modeling Framework (EMF) [Budinsky et al. 2003] is a modeling environment for the Eclipse toolset. The simplified version of the EMF metamodel is depicted in Figure 3.2.

Figure 3.2 The EMF Metamodel

\textit{EClass} is a corresponding construct to \textit{Atom} in VMTS, \textit{EAttribute} represents an attribute (it is stored in the graph labels in VMTS), and \textit{EDatatype} is a data type for an attribute (in VMTS it resides on the metamodel level). This is basically a two-layer modeling architecture.

MOF (c.f. 2.1.3) defines a four-layered architecture, but modeling environments are not among the intended usage scenarios, thus guidelines are not provided for that purpose. Net Beans Metadata Repository (MDR) [NetBeans] claims to implement MOF 1.4 structure. Therefore its metamodel is probably provided by the interfaces included in the mentioned version of the MOF standard. Its technical documentation is not complete, thus it is hard to discover how it handles the instantiation, and other features exhibited by a metamodeling environment.

The ADONIS [Fill 2004] metamodeling platform is a typical three-layered modeling environment. The metamodels in ADONIS is described in the ADONIS Library Language (ALL). The syntax of the instance layer can be specified in the ADONIS Definition Language (ADL) or XML.

3.2 Model Integrated Computing

Model Integrated Computing (MIC) [Sztipanovits & Karsai 1997] [Sztipanovits & Karsai 2002] [Karsai et al. 2003b] is a “design methodology used to create and evolve integrated, multiple-aspect models of computer-based systems using concepts, relations, and model composition principles to facilitate systems/software engineering analysis of the models and automatic synthesis of applications from the models” [Sprinkle 2004].

MIC focuses on models, supports the flexible creation of modeling environments, and helps following the changes of the models. At the same time it facilitates code generation and provides tool support for turning the created models into code artifacts. Metamodeling environments and model interpreters together form the tool support for MIC (Figure 3.3).
So far MIC is the only methodology which requires metamodeling environments and model processors, and provides a framework for them to cooperate in creating Computer-Based Systems (CBS) in the practice.

### 3.3 Model Transformation Systems

As the results implemented in VMTS have counterparts in other transformation systems, the related tools and their theoretical solutions are presented. Although there are numerous graph transformation systems enlisted below, VMTS has been most influenced by the GReAT model transformation system (causalities, parameter passing) and PROGRES (cardinality assertions).

The classification aspects that we used are the following:

- **Model specification**
  - Topological
  - Attribute
  - Constraints
- **Matching algorithm**
- **Rule specification**
  - Topological transformation
  - Attribute transformation
  - Constraints
- **Control flow**
- **Primary application area**

Other classification aspects (taxonomies) can be found in [Mens & Gorp 2005] [Mens et al. 2005].

#### 3.3.1 PROGRES

The **Programmed Graph Rewriting System** (PROGRES) [PROGRES] [Schürr 1994] [Zündorf 1996] [Reekers & Schürr 1997] is a programming environment and a language at the same time. While the language offers graph visualization and the related text information, it is also possible to obtain a view that contains the same data in textual format. PROGRES allows assigning types to nodes and edges, and
specifying inheritance between the class node types. Moreover, it is possible to define cardinality constraints on the edges, which corresponds to the UML multiplicity construct. In PROGRES only interval sets can be specified. Apart from the standard ones, the types, however, must be specified in a host programming language (preferably in C). In that sense, the PROGRES representation is unlike a UML class diagram, in which the nodes define custom types.

For graph queries and graph transformation PROGRES offers path expressions for efficient pattern specification. Apart from these expressions, patterns may contain negative edge conditions, cardinalities, and attribute conditions (counterparts of OCL constraints). PROGRES also supports optional LHS nodes, which are matched when they exist in the host graph, but the rule does not fail otherwise. In VMTS optional nodes can be matched including zero multiplicity value. Regarding the implementation of pattern matching, PROGRES provides constructs for rule firing and for sequencing the rules to form a controllable transformation process. PROGRES offers refined control structures; both imperative and declarative approaches can be used in either a deterministic or a non-deterministic manner. ACID transactions are also allowed in the control specifications. To solve the subgraph isomorphism the tool creates an action graph used to establish a valid search plan executed in an order given by special action priority heuristics [Zündorf 1993]. PROGRES supports step-by-step execution of the specifications or execution in a standalone package with user interface.

3.3.2 GReAT

The GReAT tool [Agrawal et al. 2002] [Karsai & Agrawal 2003] [Sprinkle & Karsai 2004] [Sprinkle 2003] [Sprinkle et al. 2002] [Sprinkle et al. 2003] [Karsai et al. 2003a] [Agrawal et al. 2005] provides graph transformation services complementary to the traditional GME interpreters. With GReAT it is possible to create Visual Model Processors to transform GME models, or other artifacts having UDM representation.

The general architecture and tool dependency of GReAT is depicted in Figure 3.4.

![Figure 3.4 The overall architecture of GReAT](image)

GReAT uses UDM as the underlying graph database interface to CORBA, XML or MGA backends, the latter one can also be accessed from the GME editor, thus providing a GReAT interface for VMPs.

An example for the GReAT pattern specification language is illustrated in Figure 3.5.
Figure 3.5 An example for the GReAT transformation language

The cardinality of a node (pattern object) is represented as a stereotype. Since determinism seems to be one of the main concerns considering the pattern language, the tool interprets match for the pattern on the left side of Figure 3.5 as it is depicted on the right side of the figure. Although both the type compatibility principle and the type system in GReAT conform to UML, the semantics of the pattern is specified by the described proprietary interpretation. The patterns are not instantiated by the matches according to the UML instantiation rules.

As it appears in Figure 3.6, GReAT introduces a grouping construct which repeats the contained patterns as many times as it is supplied by the stereotype of the group. The pattern interpretation extended in this way is referred to as Grouped Set Semantics (GSS). The pattern edges can also have cardinality to denote the parallel edges on the instance level.

Figure 3.6 The grouping construct in the GReAT pattern language

As far as the control structure is concerned, GReAT allows specifying an initial context for the matching to reduce the complexity of the general matching case. The pattern matcher returns all the possible matches to avoid the inherent non-determinism in the matching process. The attribute transformation is specified by a proprietary attribute mapping language, whose syntax is close to C. The LHS of the rules can contain OCL constraint to refine the pattern.

The internal control structure of GReAT can be illustrated the best with its metamodel (Figure 3.7).
A Transformation can contain RewritingRules connected by the directed sequence relationship. A RewritingRule can be a StartRule. An ObjectWrapper is a substitute for a type defined in the metamodel of the host graph or in the output metamodel. RewritingRules encompass these ObjectWrappers, and the LHS and the RHS objects can be connected by the PassAlong (Causality) relationships. The edges are wrapped by ObjectLinks. The control of a given transformation and the rules are specified by a model instantiating the control structure metamodel. The control flow can be deterministic, non-deterministic and recursive.

3.3.3 The VIATRA System

VIATRA (Visual Automated Transformations) [Varró et al. 2002] [Varró & Pataricza 2003] [Varró 2003] is a model transformation framework developed mainly for the formal dependability analysis of UML models. In VIATRA metamodeling is conceived specially: the instantiation is based on mathematical formalisms and called Visual Precise Metamodeling (VPM). VIATRA makes use of the metamodels, and the transformation language itself supports type checking, attribute conditions, negative patterns in addition to traditional pattern matching issues.

A recently implemented tool referred to as VIATRA2 is an Eclipse-based general-purpose model transformation engineering (transware) framework that will support
the entire life-cycle for the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains. The main usage scenario of VIATRA2 is depicted in Figure 3.8.

Using efficient importers and exporters, VIATRA2 is able to cooperate with an arbitrary external system, and execute the transformation with a native transformation model (plug-in), which is generated by VIATRA2. The native transformation uses the same importers and exporters that VIATRA2 does. The rule specification language is a proprietary pattern language with type information. The attribute specification is graph-based. The attribute transformation is performed by abstract state machine statements, and there is built-in support for attributes of basic Java types. The model constraints can be expressed by graph patterns with arbitrary levels of negation. The rule constraints are also specified by graph patterns. The control flow language is based on abstract state machines. The pattern matching method is CSP for the interpreter and model-specific local search for plug-ins.

### 3.3.4 Other Approaches

The sNet method [Lemesle 1998] presents the baseline of a metamodel-based model transformation. The source and destination metamodels are available and used for the rule construction. The method uses the metamodel element in the rules. It does not apply graph rewriting concepts, but traversing processor techniques in the transformation engine which takes a textual representation of the transformation code.

The Attributed Graph Grammar (AGG) [AGG] [Ehrig et al. 1999a] [Taentzer 2004] system is a Java-based visual programming environment. It also applies textual elements in the AGG language. The “visual programming” is performed by graph rewriting, taking either the DPO or SPO approach (c.f. Section 6.2), or injective (one-
to-one mapping). The pattern matching method for the LHS graphs is based on CSP (Constraint Satisfaction Problem) approaches. The tool offers graphical editing and debugging facilities. In AGG the metamodel is a simple type graph: a graph $G$ is typed over a type graph $TG$, if and only if there is a graph homomorphism (c.f. Section 5.1) between $G$ and $TG$. The attribute specification is given in Java. The model constraints are specified by visual pre- and postconditions. The constraint has a premise and a conclusion part. If the premise graph can be found, the conclusion graph must also be found. Constraints with no premise are also possible, the conclusion must always be found. The constraints in the rule is provided in Java or negative application conditions (NAC), which means that the LHS must match whereas the NAC graph (constructed from the same elements and in the same way as the LHS) is not allowed to match. The control structure of AGG is given by layers. Rules within a layer are executed arbitrarily using a random number generator. The execution order of the layers is sequential; however, a loop can be defined between the first and last layer. Primarily AGG is a graph transformation system also supporting graph grammar specifications. Based on the parallelism theorem (c.f. Section 6.2.1), AGG is able to detect the possibility of non-confluent rules and can check sufficient conditions on the termination of the transformation. The GenGed (Generation of Graphical Environments for Design) [Ehrig et al. 1999a] [Bardohl et al. 2002] environment is suitable for creating visual language definitions. It is rather presentation oriented: instead of metamodeling, it specifies graphical symbols, constraints and their connections; from this information graph rewriting rules (Alphabet Rules) are generated, which serving as the graph grammar used to parse the visual language. GenGed uses AGG as the internal graph transformation engine; thus, we restrict our categorization to AGG. For the editing features, a graphical editor is also generated to support the newly created visual languages. GenGed has been replaced by a new project called Tiger (Transformation-Based Generation of Modeling Environments) [Ehrig et al. 2005c] recently, which generates an Eclipse-based editor. Tiger also uses the AGG transformation engine.

The FUJABA tool (From UML to Java and Back Again) [FUJABA] uses UML class and behavioral diagrams to produce Java code, in order to provide an equally formal and practical system design and specification language. Reverse engineering functionalities including design pattern recognition are also available. The FUJABA’s metamodel can be extended by inheritance instead of instantiation. In this way FUJABA has only two layers, both of them are editable. For the created model the tool is capable of providing plug-in interfaces, which are the corresponding constructs to Traversing Model Processors. The rule patterns in FUJABA are expressed in story diagrams [Fisher et al. 1998], which is an integrated language for specifying rewriting rules, attribute transformation and control flow. The pattern matching optimization strategies are the same as in PROGRES, except for the backtracking mechanisms implemented for a decision, when multiple choices are available. Contrary to PROGRES, FUJABA does not revise the decisions made when multiple matches occur.

The transformation and simulation tool ATOM3 (A Tool for Multi-formalism and Meta-Modelling)[ATOM3] [Lara et al. 2004] uses model transformation to simulation traces in order to simulate the operations, for instance, state variable trajectories in case of continuous system simulation. It supports multilevel metamodeling supported through a model interpreting approach similar to that of GME. The system was implemented in Python, thus the attribute specification and the model constraints can be given in Python along with the attribute transformation. The rule constraints can
contain generalized negative application conditions. The pattern matching algorithm is taken from [Dörr 1995]. The model constraints are checked on events. The constraints can be pre- and postconditions to events. Constraints can be both semantic and graphical constraints. Semantic constraints are related to the valid instantiation, graphical constraints are connected with the graphical properties of the shapes (line color). Similarly to AGG, the control flow consists of layers; the rules are sequenced by priority numbers within the layers. A rule is executed only once, but in case of non-overlapping matches, the rules are applied to all the matches.

Other approaches to model transformations can be found in [Akehurst 2000] [Milicev 2002] [Sendall 2003].

3.4 The Visual Modeling and Transformation System

There are several assumptions (e.g. the availability of the metamodels) in the formal model discussed in Chapters 5 and 6, whose relevance can only be conceived via practical applications. This section is designed to outline the background and give a motivation for the theoretical results. Recently, highly configurable metamodeling environments and graph transformation techniques have successfully been applied in software system modeling and other areas. A unified treatment of these two methods is illustrated by a tool called Visual Modeling and Transformation System [Levendovszky et al. 2004a] [Levendovszky et al. 2004b] [Levendovszky et al. 2004c] [Lengyel et al. 2004c]. The concept of an n-layer metamodeling environment is outlined with the related topological and attribute issues. Built on metamodeling techniques, two alternatives for model transformation are elaborated: the traversal and the graph-rewriting approaches are discussed. In our implementation all of the aforementioned mechanisms use metamodels as a common background and as a uniform basis for storing, creating and transforming visual languages.

3.4.1 Design Objectives

The general requirements for a highly configurable metamodeling tool supporting UML, and a general purpose model transformation system are the following:

- Support for an extensible visual language including shapes and the related editor capabilities for the mainstream languages
- Elegant multilayer metamodeling solutions
- Powerful constructs for model transformation applications

The mainstream modeling languages vary extremely in shapes and editorial concepts. In UML 2.0 several shapes are required to contain other shapes visually (e.g. states), or have multiple “grid-like swimlanes” (activities). In the case of feature models, arcs must be drawn around the line ends depending on the meaning of the actual relationship. To support all these features on demand is not a negligible software engineering feat. However, if such a construct is ready and well-designed, the solution will be reusable across several applications in the future. Our tool offers the VMTS Presentation Framework, which both solves the issues above, and provides the necessary base classes. Hence, the developer needs to concentrate on the special properties and requirements of the given language when constructing a support for a new visual language, instead of implementing everything from scratch. This technique accompanies to metamodeling, and is the basis of visual plug-in development.

The metamodeling solutions constitute the basis of such a system. The main issues are the following:

- Simplicity
The simplest metamodeling system uses only one instantiation concept during the instantiation. Another closely related issue is that of layer transparency. This means that each layer is handled by the same functions within the tool. These two requirements help to make the tools and the code generators layer-independent, which eliminates a large amount of conditional logic in the code. Expressiveness is important, when it comes to modeling UML 2.0. The metamodeling language should provide enough vocabulary and instantiation power to realize the UML 2.0 MetaClass-Class-Object diagram chain being the most difficult part of the UML from the metamodeling point of view. Natural code generation support helps to reach the model elements of the transformation system as if it had been objects in an OO programming language. The metamodel is interpreted as a class diagram by the code generator, then, the tool exposes its model elements as objects of the classes generated from the metamodel. The well-defined property is ensured by treating the metamodel of a UML class diagram. In VMTS this is the main idea of the Traversing Processors.

One of the main differences between a model transformation system and a graph transformation system is that model transformation systems must provide more complex construct with regard to
- Rule specifications
- Attribute transformations
- Control flow

This is because models (especially the UML models) are capable of expressing more complex semantic constructs than graphs, which are abstract mathematical objects. Therefore, the transformation rules are required to express more complex transformations, which are given in a natural language in several cases. Thus a model transformation system should provide the rules with specific constructs like `forAll`, constraints, and attribute transformations. It is important that developers already familiar with UML be able to transfer their knowledge to specify rules. In VMTS, rules are based on UML class diagrams and the standard instantiation relationship between UML class and object diagrams. However, in the case of attribute transformation, usually a designated language is used (as of writing, Java is the most common). A general solution is desirable, where attributes and the attribute transformations are independent from a programming environment. In VMTS, attributes are represented in XML, and attribute transformations are XSLT scripts, which can be regarded as a tree transformation from a theoretical point of view. The control flow is a key point in a model transformation system. As opposed to graph grammar tools, model transformation requires explicit and not random-based constructs, since they help to design a terminating transformation which works correctly in practice. Thus, intuitive control structures are required. Since the rules are not independent any more and their execution order is predictable, parameter passing can increase the efficiency of the rule execution. The control flow in VMTS is specified as a stereotyped UML activity diagram with parameter passing facilities between the rule elements.

Since we want to design a general purpose model transformation system, there are factors whose influence should be taken into consideration.
- Performance
- Moderate development effort
There are cases, especially in regards to the tools used for verification, where the performance is not focused, because the time requirement of a verification tool used on the target model is much higher than that of any transformation tool with reasonable characteristics. If performance is critical the best implementation language would have been C or C++. However, we decided to use a higher-level programming language, since VMTS was a prototype to check our novel ideas, and it would have not been practical if the mistakes in the theoretical concepts are revealed after a significant coding activity. Taking the algorithmic background into consideration, the performance factors were considered when designing VMTS. The heuristics can be requested explicitly by the user (e.g. specifying pivot elements for the pattern matching), thus these heuristics do improve the performance in practice. So far it has been worked out on the theoretical level, measurements will take place in the not too distant future. The main reason for this is that VMTS has been applied in medium-sized models, huge models (usually coming from reengineered program code) have not been needed up to now in our practical applications.

VMTS is a proof-of-concept implementation for the uniform approach to metamodel-based storage and transformation. UML compliance is an important feature to ensure the practical applicability. The basic formalism used by VMTS is labeled directed graphs. This construct serves as the formal background, and it is capable of maintaining the mathematical relations directly. VMTS applies standard technologies like XML and enforces the separation of concerns: the presentation, storage, and validation modules have been separated as clearly as possible. In fact, VMTS has been implemented in .NET using C#, however, language and environment independent principles are focused in this work; only XML support and an object-oriented language are assumed.

3.4.2 Overview

This section presents the principles of a model storage and transformation software package called Visual Modeling and Transformation System (VMTS). VMTS illustrates an approach uniformly treating model storage and model transformation. What links model storage and transformation together is the notion of the metamodel. Modeling environments built on metamodeling are highly configurable (visual) modeling tools allowing constraints to be specified in advance. Model transformations can be used for model and code generation as well as for model modifications. One of the most promising directions is to create general transformation systems. The most used UML processors need to incorporate rich semantic information to specify the transformation rules. This is because currently the standard UML has semantics in plain English, which cannot be formalized to help the rule formulation. As it is illustrated in VMTS, metamodeling can be the basis of different model transformation methods as well.

This section is devoted to the concise discussion of concepts without delineating the implementation details. For additional information (constraint handling, implementation details, comparison to MOF, UML metamodels) please refer to [VMTS].

The VMTS system architecture is depicted in Figure 3.9. The user interface (Adaptive Modeler) is functionally separated from the model storage unit (AGSI Core, Attributed Graph Supporting Inheritance), which uses an RDBMS (Relational Database Management System) to store the model information. The model transformation can be accomplished by (i) traversing processors, (ii) the rewriting engine or (iii) other applications.
Figure 3.9 The VMTS System Architecture

Traversing processors walk through the model graph and create other artifacts either in AGSI or outside it. “Classical” code generators usually fall into this category: by analyzing models they produce code in a specific programming language. The Rewriting Engine executes metamodel-based visual model processors created in the Rule Editor. The AGSI Core exposes its interface to any other applications which may use other techniques to process AGSI data.

3.4.3 An n-layer metamodeling environment

Modeling environments have to face the challenge of following up version updates (even in standard UML, e.g. compare UML v1.4 and v2.0, Section 2.1.1) and the varieties of models (UML sequence diagram, class diagram and feature models). To save development efforts meeting these requirements, VMTS uses metamodels to create a flexible, visually configurable modeling environment. Metamodeling is based on the instantiation relationship i.e. the relationship between the UML class diagram elements and UML object diagram elements. It means that VMTS uses a simplified UML class diagram as a metamodel language to define models (e.g. UML statechart diagram, UML use case diagram and UML class diagram). If the UML class diagram is instantiated there are three layers involved: the UML object diagram, the UML class diagram and the metamodel of the UML class diagram. In addition, there are two more layers in VMTS: (i) the read-only meta-meta model which specifies the metamodeling language and (ii) the one in the internal structure (this is a labeled directed graph). The model storage part of VMTS is called Attributed Graph Architecture Supporting Inheritance (AGSI). AGSI layers are designed so that every model can be a metamodel for other models, but the five layers discussed above have turned out to be enough in practical scenarios.

3.4.3.1 Topological Considerations

AGSI can handle graphs via three basic constructs: (i) nodes, (ii) directed edges and (iii) labels assigned to nodes and edges. So far it would be a storage structure for directed labeled graphs, thus metamodeling support needs to be added. Each node and edge holds a bidirectional connection to other nodes and edges, respectively. This is the type-instance mapping. These mappings must not form a loop so that graphs can be organized into tiers, each of which corresponds to a modeling layer.

Although this structure is capable of storing model topology, attributes and appearance information in the labels, modern modeling languages, and environments (including UML) require additional notions. (i) Models should be traversed via a containment hierarchy: every node has a unique parent which contains it possibly with
several other elements. Basically this construct is more suitable for easier traversal and better-structured storing, than displaying the model in a tree view. Consequently, this structure is not a concept supporting visual presentation, even if the presentation benefits from it. For containment hierarchies AGSI maintains a parent-child bidirectional mapping. (ii) Inheritance support (a directed mapping from the descendants to the ancestors which must not contain loops) is a natural requirement for every modeling system even in the metamodel levels. (iii) Association classes are strange constructs, because a class (association class) is connected to the middle of an edge (association). Although this arrangement could be resolved by inserting a pseudo-node with no semantic meaning, it would be better suited to be stored in a regular graph structure. For conceptual reasons we decided to add support for this model element as well. All the aforementioned features have hard-wired support in AGSI. Furthermore, inheritance needs special treatment from the topological angle: descendant types inherit the relationship types from their ancestor types, meaning that instances of the descendant types can be adjacent to instances of the types adjacent to the ancestor type as well.

The read-only root metamodel of AGSI is depicted in Figure 3.10.

![Figure 3.10 The read-only root metamodel of AGSI](image)

_SystemNode_ and _SystemRelationship_ are provided by AGSI. These are hard-wired constructs that correspond to the node and the directed edge elements of the labeled directed graph model. After specifying the behavior of these special features, it should be modeled visually so that hard-wired mechanisms must be applied to certain elements instead of instantiating them in the regular way. Taking inheritance as an example, this works as follows: Since this relationship is a natural construct of AGSI, it can be introduced at any level. When one wants to have an inheritance relationship in model A, a relationship with <<System MetaInheritance>> should be introduced on the Am metamodel level. On level a (the instance of model A) the relationship does not appear, and the hard-wired inheritance mechanisms are executed. For example to have an inheritance in a UML class diagram, we specify <<System MetaInheritance>> between metaclasses. Thus, we are able to use generalization between classes in a UML class diagram and in the UML object diagram we are able to draw links instantiating the inherited associations. As inheritance is a hard-wired construct, it is available on the read-only root metamodel level as well. Since inheritance is a useful construct on editing metamodels, the read-only root metamodel specifies a <<System MetaInheritance>> relationship to enrich the metamodel editing facilities. This on-demand introduction of inheritance is not required in
metamodeling environments that offer two layers for editing and have a root metamodel hard-wired, because if we draw a class metamodel, we are able to edit class models but not object diagrams. Containments and association classes are implemented similarly: relationships with stereotype `<<System MetaContainment>>` and `<<System MetaAssociationClassRelationship>>` are the metaelements for these constructs. In a sense AGSI is not a strict metamodeling environment, because `System Relationship` offered by the read-only root metamodel layer can appear on any level. We restrict this facility on the user interface layer to avoid confusion. This construct can be also used for temporary relationships between objects of any type during a transformation process.

3.4.3.2 Attribute Issues

In the labeled directed graph model attributes are stored in the labels. In AGSI, the labels are XML files, and the attributes are represented in an XMI like format [Mezei et al. 2005a] [Mezei et al. 2005b]. If the attribute data structures are instantiated, there are two types of attributes: (i) metaattributes, which can be instantiated, and (ii) attributes residing in the metamodel elements, but not appearing in the instantiated model elements. Typical examples are attributes and methods in UML classes. Attributes (e.g. “x : integer”) are instantiated (e.g. “x = 10”), but methods (e.g. “f(x : integer) : bool”) do not appear in the corresponding objects. In AGSI the metaattributes are converted to an XSD file, which is the schema for the XML file storing the attributes on the instance level. This method is quite flexible: adding, removing, and changing attributes require altering an XML file, which can easily be processed by tools in any modern programming environment. Figure 3.11 illustrates this method for the UML metaclass-class-object instantiation chain.

![Figure 3.11 The attribute instantiation chain](image)

In addition, there are attribute-related issues of the features discussed in the previous section. The descendants inherit the attributes of the ancestors, thus the XML files should be merged along the hierarchy such that the root element has the lowest priority and the lowest descendant has the highest priority, provided that an attribute with the same name is specified in more than one element in the inheritance hierarchy path. Abstract nodes (abstract is a hard-wired attribute) cannot be instantiated. Since edges `SystemRelationship` can have attributes as well, AGSI does not need association classes to specify metaattributes for edges, however, this facility is
supported as well. The attributes stored in the edges have higher priority than the ones residing in the association node when they are merged.

3.4.3.3 Constraints and instantiation

A general graph storage system is able to store all kinds of graphs. As far as metamodeling environments are concerned, the models should be restricted by the rules of the instantiation. Type constraints should be enforced when edges are created. This involves verifying multiplicities on the association ends. Other constraints must also be considered: abstract classes should be prevented from being instantiated. Moreover, model elements can contain constraints expressed in the Object Constraint Language (OCL). AGSI treats these constraints uniformly via a general interface. When a new element is to be added, these constraints are validated by \texttt{AGSIInstantiationConstraintModul} and \texttt{AGSIOCLValidator}. The OCL validator \cite{Lengyel05b} \cite{Lengyel05e} \cite{Lengyel05g} is a native code module compiled from the metamodel on the first instantiation. OCL constraints can have error messages in the form of special comments; if a constraint is violated, this message is reported to the end-user.

3.4.3.4 Adaptive Presentation Facilities

There are two important design considerations related to the presentation of the models: (i) the model representation and user interface (UI) concerns should be clearly separated, and (ii) every model type (including new ones) should be displayed, including UML class diagrams, UML sequence diagrams, feature models etc.

To enforce the separation, each AGSI element has an XML file where the UI module can store proprietary information. The UI is a separate application (Adaptive Modeler \cite{Angyal04} \cite{Levendovszky04d}, Figure 3.1), which uses AGSI via a well-defined set of Façades. Adaptive Modeler has the responsibility to display the particular AGSI constructs, to map the changes from the UI back to AGSI and to report any errors.

To achieve arbitrary model support Adaptive Modeler contains a framework and a generator for creating plug-ins. Plug-ins can be developed separately and registered to display a model type (e.g. sequence diagram plug-in). The framework is built on the Model-View-Controller design pattern and has several predefined shapes to help in efficient plug-in development. A plug-in is assigned to one or more metamodels to display their instance diagrams. If a metamodel is not assigned to any plug-in, a default plug-in can be used for editing.

There are two metamodel-independent parts in Adaptive Modeler: (i) an attribute panel displays the attributes from the XML documents assigned to the AGSI elements and (ii) a tree view, which presents the model elements in the containment hierarchy defined by the \texttt{<<SystemContainment>>} relationships between the model elements. They are metamodel-independent in a sense that they work for models of arbitrary metamodels. These components obtain the necessary information from the actual metamodel.

3.4.4 Model Transformations

There are two important categories of transforming a labeled directed graph. The simplest and the most universal one is the traversal approach, whereas graph rewriting can be used as a transformation mechanism. Model transformation means converting an input model that is available at the beginning of the transformation process to an output model. As it was discussed in Section 2.2.1, Model Driven Architecture [OMG
MDA Guide] sets out a more restrictive definition: the output model should describe the same system as the input model. As VMTS has been designed to be able to specify not only MDA model compilers, but more general transformations, we omit this caveat of MDA from our definition.

3.4.4.1 Traversing Model Processors

The simplest method to transform models is to traverse them using a specific programming language, while changing the appropriate parts of the input models or producing an output model. Traversing Model Processors (TMP) offering this approach usually use five basic graph operations: (i) create node, (ii) connect nodes, (iii) delete node, (iv) delete edge and (v) set label. Of course these operations may vary, but these categories can usually be observed. As far as model transformations are concerned, there are usually specific parameters of these operations, for instance, node creation requires type information of the node to be created, and set label usually supports attributes.

In VMTS traversing processors, the models and their elements are regular objects in an object oriented programming language. The creation of a node having a specific type is the creation of an object having the corresponding type; deletion of a node means destroying the related object; the attributes can be set via member variables named after the corresponding attribute names. The deletion of an edge means removing a reference from one object to another. Edge creation can be performed by adding a reference to the target object in the appropriate array member of the source object, or vice versa if it is a bidirectional edge.

In VMTS the TMPs can be either AGSI-aware or AGSI-unaware. AGSI-aware traversing processors maintain the connection with the storing system: the execution of the traversing processor starts with loading the attached model from the model storage, and any changes made in the programming language are propagated back to AGSI. In case of AGSI-unaware TMPs, the changes made in the traversing processor do not affect the input model. In order to generate a model processor in an object oriented programming language, two model levels must be considered: (i) the current layer, which we want to traverse (these will appear as objects in the programming language) and (ii) the metamodel layer of the current layer, because that provides the class definitions for the objects.

In case of AGSI-aware model processors, the connection with AGSI is maintained via the AGSI Interface component. In both AGSI-aware and unaware cases an object container is created, which holds references to the objects loaded from AGSI. In this way traversing and querying the objects are convenient for the programmer. For each metamodel a generator (TraversingProcessorWizard) creates the framework and the metamodel-specific code automatically, so that the programmer can traverse the model by the object container or by variables (references) named after the name attribute of the model element (if the name of a use case is uc in the model, an object is created with the name uc instantiating the UseCase class). Model processing code generators are special TMPs. In practice, they are used most often. In VMTS the framework generation for a traversing processor always includes a code generation part: the metamodel layer is regarded as a UML class diagram, and classes are generated from it.
3.4.4.2 Visual Model Processors

Visual Model Processors (VMP) do not replace TMPs, instead, they provide a visual alternative way of model transformation. In VMTS VMPs use graph rewriting as the transformation technique. Firing a graph rewriting production rule consists of three steps. (i) Finding an occurrence of the LHS in the host graph. This means obtaining a subgraph of the host graph that is isomorphic to LHS. This subgraph is called match. (ii) Removing the nodes and edges from the redex graph which are in the LHS but not in RHS. (iii) Gluing the nodes and edges to the redex which are in RHS, but not in LHS. This technique has been used successfully in several transformation systems, some of them are pointed out in Chapter 8. In model transformation systems the host graph is the input model of a transformation step (LHS, RHS pairs), where LHS consists of input model elements. VMTS, however, takes a novel approach introduced in [Levendovszky et al. 2002], which specifies the rewriting rules (LHS and RHS) in terms of the metamodel.

Consequently, instead of finding a direct occurrence of LHS, a part of the input model must be found which instantiates LHS. In this chapter we do not consider the sequencing of the transformation steps but examine one step. In this simplified approach, we call the input model the model which the rule is applied to and the output model is the result of the transformation step.

3.4.4.2.1 Topological Considerations

In VMTS, LHS and RHS are defined using UML class diagram syntax. The constructs that form a metamodel-based LHS specification are inheritance and multiplicity support (Figure 3.12). Inheritance support is analogous to the natural type compatibility of OO languages: the derived class can always be passed where the ancestor class is expected. This means that a class element in LHS always matches its descendant types in the input model. That facilitates generalization in the rules as well as abstract types. Multiplicity support is accomplished by allowing multiplicity values on the association ends. The most important differences between the LHS and a metamodel are the following: (i) LHS is required to be connected (i.e. a path must exist from any class to any class ignoring the association types), (ii) the match found for LHS is maximal in a sense that the actual matched multiplicity value (the number of links matched to the association) is the greatest possible value from the specified multiplicity interval, which do not contradict any other part of the match. Multiplicities are local constructs, so all the matches in the input model are not found and compared to achieve the greatest actual multiplicity; it only means that no more links can be added to the match which is also a match for a particular LHS. Chapter 4 is devoted to this issue in detail.

3.4.4.2.2 Attribute Issues

The semantics for LHS has been described in the previous section. Processing RHS (firing the rule), however, is related to the attribute transformation issues: if an edge in RHS has an exact multiplicity, then the related object must be filled with attributes; if the multiplicity value is not determined uniquely (like *), the number of the created attribute sets determines the objects to be created. The attribute transformation is accomplished by Extensible Stylesheet Language Transformation (XSLT) scripts.
Figure 3.12 Rule specification in AGSI

LHS elements can have causality relationships pointing to RHS elements. Causalities can express the modification or removal of an LHS element, and the creation of an RHS element. XSLT scripts can access the attributes of the object matched to LHS elements, and produce a set of attributes for the RHS element to which the causality points. This method was successfully applied in code generation from class diagram with VMP (Section 8.1.2), and the idea can be extended in a straightforward way (even to a more sophisticated language than XSLT) when it is needed.

The most important thing to be preserved is the conformance to the formal background, i.e. the labeled directed graph structure. Hence the attributes must belong to the label, which is a string expression (usually defined by a set of characters, where the elements of the possible sequences can be taken from). From the practical side an XML representation seems to be a convenient choice. This solution, however, must fit the other concepts: (i) the attributes in the metamodel level should specify the attributes in the model level (the XML document of the metamodel element needs to be transformed into an XSD (XML Schema Definition) schema), (ii) during the transformation steps (rewriting rules) the attribute transformation scripts must be specified and evaluated. The natural and widely adopted way of transforming XML documents is using XSLT. In VMTS both the metamodel level-model level relationship and the attribute transformations are addressed by XSLT.

In creating a rewriting rule, we specify causality relationships from the elements of the LHS rule to the elements of RHS rule. An RHS element must always have at least one causality arrow pointing to it. The XSL rules are assigned to the causality relationships and specify what transformation should be executed to create the attribute set of the RHS element. Causality relationships can be one of the following types: (i) creation, (ii) modification or (iii) deletion. Creation results in creating one or more elements of the type specified by the RHS element. Because of the metamodel-based rule specification the cardinality of the element to be created can be characterized by “*” multiplicity, which means the “as many as possible” cardinality constraint. Consequently, the attribute transformation affects the topological aspects as well: the number of the resulting attribute sets defines the number of output nodes.
to be created. The second type of the causality relationship (modification) describes the modification of the existing elements; these are attribute transformations only. Finally, the deletion causality denotes the deletion of the selected LHS node.

When a transformation step is executed (assuming that the matches of the LHS elements have already been found) we traverse LHS and for each element (i) the type of the causality is found, if any (ii) if the causality type is creation, the type of the new element is retrieved from RHS (iii) then the execution of the XSL rules takes place (iv) finally, the new elements are created, filled with the attributes, and the connections are established based on the topologic rule specification. If an LHS element is tied to a modification causality arrow, only an XSL rule is performed on the element matched to the LHS node.

In Chapter 8 we illustrate the steps discussed above and give more details on the attribute transformation technique by case studies.

3.4.4.2.3 Constraints in the Transformation Rules

Specifying constraints is not among the focused topics of this thesis; however, the basic issues are pointed out to present the full palette of the services provided by VMTS. This work is the subject of an ongoing research with László Lengyel, having promising results [Lengyel & Levendovszky 2004] [Lengyel et al. 2004a] [Lengyel & Levendovszky 2005] [Lengyel et al. 2005a] [Lengyel et al. 2005b] [Lengyel et al. 2005c] [Lengyel et al. 2005d] [Lengyel et al. 2005e] [Lengyel et al. 2005f] [Lengyel et al. 2005g] [Lengyel et al. 2005h] [Lengyel et al. 2005i]. The metamodel-based specification of the rules allows assigning OCL constraints to the rules, using the guidelines of the UML standard [OMG UML 2.0]. Because these constraints are bound to the rules, they are able to express local constraints (because the elements not appearing in LHS or RHS cannot be included in the OCL statements). Although the specification has this local nature, it does not mean that validating them ignores checking other model elements in the input model: constraint propagation needs to be taken into account by both the algorithmic background and the user of the transformation on specifying the constraint.

Constraints specified in LHS can be considered as a precondition of the rule: if it does not hold, the transformation step fails to start. It can be thought of as a refinement facility of the definition of pattern matching. Constraints bound to the RHS elements express a postcondition of the rule: if they are not satisfied, the match cannot be completed. In other words if the rules are performed successfully, the constraints in RHS are satisfied.

Apart from these explicit guarantees, there can be other, implicit ones. E.g. it is assumed that an LHS constraint \( C \) is true, and the rule never changes the properties related to \( C \). In this case if we enlisted \( C \) among the postconditions, we would not change the behavior of the system i.e. any sequence of the rules exhibit the same result in either case.

This easy and standard way of specifying constraints also stems from the application of the metamodel similar to the UML class diagram to specify the transformation rules.
3.5 Chapter Summary

3.5.1 Related Work

Metamodeling environments can be broken down into categories based on the following properties (Table 3.1):

- Number of supported layers (not counting the hard-wired root metamodel)
- Layer transparency: whether the same functions operate on each layer

<table>
<thead>
<tr>
<th>Layers</th>
<th>Layer-Transparent</th>
<th>Not Layer-Transparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>UDM</td>
<td>GME, ADONIS, EMF,</td>
</tr>
<tr>
<td>&gt;2</td>
<td>VMTS</td>
<td>MDR, AToM²</td>
</tr>
</tbody>
</table>

Table 3.1 A comparison of metamodeling tools

Among the graph transformation environments it is impossible to make a fair comparison, since their purpose, implementation platforms, usage and concepts are so different. There are only a few environments (for example, GReAT, VIATRA, VMTS) which are integrated into a metamodeling environment. The model transformation techniques can also take the advantage of the metamodeling concepts, but none of the tools above use metamodel language to specify the graphs of the rewriting rules.

Similarly to the tools, graph transformation methods are also hard to be compared. A summary of the discussed transformation systems in table format can be found in Appendix B. The generality and the enforcement of the identity condition made the DPO approach suitable for model transformation and the target of our choice. The context free grammars are not expressive enough for a transformation system, where its advantages in the field of parsing turn to be a disadvantage in the area of model transformation systems.

3.5.2 VMTS Summary

This chapter has presented the concepts of a system providing model storage and transformation facilities. What brings together these techniques is the extensive use of metamodels. The fundamental structure of AGS I is basically a labeled directed graph which lends itself easily to theoretical proofs and mathematical description. The flexibility and the configurability of the storage system have been achieved by metamodeling. It has been illustrated that a modeling environment built on metamodeling techniques needs a few hard-wired constructs: node, edge, labels (practically in XML format), inheritance, containment and association classes. The existence of the metamodel facilitates mapping the model elements to any object oriented programming language in case of generating traversing model processors.

Visual Model Processors use metamodel elements in the rule specification, ensuring a uniform treatment of powerful novel constructs and those applied in other transformation systems. Due to their overlapping, but not equivalent application areas, visual and traversing model processors should co-exist as appropriate tools for different objectives. It has been shown that one type of instantiation (the one between UML class and object diagram elements) is enough to build an n-layered metamodeling environment (as opposed to MOF, c. f. Section 2.1.3), and metamodels can be specified in a way suitable for all of the discussed purposes. If a developer is familiar with UML (class diagram, object diagram, OCL), it is really easy to use the same concepts and language for describing model transformations as well.
3.5.3 Open Issues in VMTS

In this chapter the main results have been summarized with the model transformation approaches. Results implemented in VMTS constitute the baseline for the contributions included in this thesis. The fusion of metamodeling and model transformation system provides the applications and the motivations for the theoretical results. The subgraph isomorphism serves as a basis for the metamodel-based transformation rules introduced in this chapter. Introducing the `typeof` relation as a homomorphic mapping between the model and the metamodel elements helps proving useful properties for compatible or partially compatible metamodels, novel conditions can be specified for the offline validation of metamodel-based model transformation rules. Extending the DPO approach for metamodels facilitates sufficient conditions for the parallel executions of the metamodel-based rewriting rules.

Therefore, the next chapters propose solutions for the open issues having arisen so far along with their proven solutions. The goal is to create theoretical results which give existence to a transformation system introduced in Chapter 3. There are problems that still remain to be solved.

- Create an algorithm which executes metamodel-based transformation rules
- Create a formalism which serves as the basis for topological validation of the transformation rules
- Extend the existing mathematical results related to the parallel execution (DPO approach) to rules containing metamodels

Accordingly, the algorithmic background is developed in Chapter 4, the formalization of the instantiation relationship is discussed in Chapter 5, while Chapter 6 is devoted to the extension of the DPO approach.
4. The Algorithmic Background

To develop the algorithmic background for a metamodel-based LHS that must be instantiated, the properties of the UML instantiation must be studied. After finding the correspondence between the number of objects and the multiplicity values, we develop an algorithm that is able to execute the transformation rules specified in terms of the metamodel elements. In this section, we use graph theoretical notation, which is built on set theory. We have chosen this model because the mathematical basis (subgraph isomorphism) uses graph theory with different graph representations. The closely related publications are [Levendovszky et al. 2002] [Levendovszky & Charaf 2005] [Levendovszky et al. 2005c].

4.1 Pattern Matching and Subgraph Isomorphism

Pattern matching lies at the heart of the model transformation systems that use graph rewriting techniques as the core mechanism of the transformation engine. Traditional algorithms take into account topological considerations only, ignoring type and attribute constraints. A graph rewriting-based transformation is a sequence of rewriting rules [Rozenberg 1997] that contain a left hand side graph (LHS) and a right hand side graph (RHS). The graph which the rule is applied to is referred to as a host graph. Applying a rewriting rule means to find a subgraph in the host graph which is isomorphic to LHS.

The host graph $G^H(V^H,E^H)$ and the LHS graph $G^{LHS}(V^{LHS},E^{LHS})$ are given by their edge (E) and vertex (V) sets. Considering models $E^{LHS}$ is modeled as a multiset. The labeling functions $h^H : V^H \mapsto \Omega$, $l^H : E^H \mapsto \Omega$ assign the appropriate set to an arbitrary alphabet ($\Omega$).

As far as graphs are considered, subgraph isomorphism and pattern matching are equivalent, but considering, software models there can be more complex constructions in pattern matching. There are two main approaches to the subgraph isomorphism problem: (i) search with local heuristics, and (ii) perceiving it as a constraint satisfaction problem (CSP). Because the high-level algorithms described in [Levendovszky et al. 2002] have strong local nature, we have decided to apply the concept of the algorithms based on local heuristics.

In the remainder of this chapter, the algorithmic achievements related to the new results discussed.

4.1.1 Ullmann’s Algorithm

Obviously, the subgraph isomorphism problem can be solved by performing a brute force search over the search space. For instance, [Ullmann 1976] and [Terék 2002] describe a naïve brute force method, which serves as a basis for the Ullmann’s algorithm. This approach uses an adjacency matrix ($A^H$) as its central data representation. Based on these notions, an algorithm for the subgraph isomorphism problem can be stated rather informally as follows. All permutation of the $A^H$ matrix should be taken such that the graph represented by an arbitrary $A^{HP}$ permutation as an adjacency matrix must be isomorphic to $G^H$. Then, each $A^{HPS}$ submatrix of each $A^{HP}$ permutation with the same dimension as $A^{LHS}$ should be compared to $A^{LHS}$. If each $a_{ij}^{HPS}$ element of $A^{HPS}$ are greater than or equal to the corresponding $a_{ij}^{LHS}$.
element of $A^{LHS}$, the subgraph represented by $A^{HPS}$ contains a subgraph isomorphic to $G^{LHS}$.

To produce the aforementioned permutations and their submatrices, a special $P$ permutation matrix can be applied. This $P$ matrix contains exactly one (1) value in each row, and no column contains more than one (1) value. If an arbitrary $X$ matrix is left multiplied by such a $P$ matrix, the result is the $X$ matrix with rows permuted. If column $k$ contains no one (1) element in $P$, the $k$th row of $X$ is omitted from the resulting matrix. If the rows are permuted, columns must be permuted in the same way to maintain the isomorphism to the graph represented by the original (not permuted) adjacency matrix. This is achieved by right multiplying the result with $P^T$ (T denotes transposition), which has a similar effect on the columns as left multiplication with $P$ on the rows. If $p_{ij} = 1$, row $j$ becomes row $i$, and the columns are permuted in the same way. It means that $P$ maps node $j$ of $G^H$ to node $i$ of $G^{LHS}$. These steps are illustrated in Figure 4.1.

![Figure 4.1](image)

(a) Permuting rows. (b) Permuting the corresponding columns. (c) Comparing the result to $A^{LHS}$.

Consequently, if all possible $P$ matrices are generated and the elaborated steps are taken for each $P$, all subgraphs of $G^H$ that are isomorphic to $G^{LHS}$ can be found. A pseudo code for this algorithm is depicted on Figure 4.2.

```
MATCH( matrix P, int j): bool
1 if j = n then
2    if $a^H_{ij} \geq a^{LHS}_{ij}, \forall i, j$ then return true
3 else return false
4 end if
5 for i ← 0 to m − 1
6    $P[i, j] ← 1$
7 if MATCH(P, j+1) return true
8 $P[i, j] ← 0$
9 end for
10 return false
```

![Figure 4.2](image)

A simple match algorithm

Using the pseudo code detailed above as a basis we present an algorithm proposed by Ullmann which is a well-known and commonly used subgraph isomorphism
algorithm [Ullmann 1976]. This algorithm also operates on adjacency matrices; in addition, it maintains a matrix \( M = [m_{ij}] \), which is defined as follows:

\[
m_{ij} = \begin{cases} 
1, & \text{if the degree of the node } j \text{ of } G^H \text{ is greater than or equal to the degree of the node } i \text{ of } G^{LHS} \\
0, & \text{otherwise.}
\end{cases}
\]

If an \( m_{ij} \) element is zero, the node \( i \) of \( G^{LHS} \) cannot be mapped to the node \( j \) of \( G^H \).

The most significant contribution is the refinement procedure, which enforces arc consistency [Terék 2002] whenever a new mapping is created between a \( G^{LHS} \) node and a \( G^H \) node (i.e. an element of \( P \) is set to 1). Then, the procedure maintains the matrix \( M \) by checking each of its element against the following condition:

\[
(\forall x)(\exists y)(a_{ix}^{LHS} = e) \Rightarrow (\exists y)(a_{ij}^H m_{xy} \geq e)
\]  

(4.1.1)

This constraint expresses that if the node \( i \) of \( G^H \), adjacent to the node \( x \), is mapped to the node \( j \) of \( G^{LHS} \), there must exist a node \( y \) of \( G^{LHS} \) which is adjacent to the node \( j \), and \( m_{xy} \) allows the mapping. If Eq. (4.4.1) is not satisfied, \( m_{ij} \) is set to zero.

```
MATCH(matrix P, int j): bool
1 if j = n then return true
2 for i ← 0 to m − 1
3 if M[i, j] = 1 then
4 M'← M
5 P[i, j] ← 1
6 for each x ≠ i M'[x, j] ← 0
7 for each y ≠ j M'[i, y] ← 0
8 M' = REFINEMENT(P, M)
9 if M'[i, j] AND MATCH(P, M' j+1) then return true
10 P[i, j] ← 0
11 end if
12 end for
13 return false
```

```
REFINEMENT(matrix P, matrix M): matrix
1 hasBeenChange←false
2 do
3 for each i, j nodes where \( m_{ij} = 1 \)
4 for each node x in \( G^{LHS} \)
5 if not \( \exists y, a_{ix}^{LHS} \leq a_{ij}^H m_{xy} \) then
6 \( m_{ij} \leftarrow 0 \)
7 hasBeenChange←true
8 end if
9 end for
10 end while hasBeenChange
11 return M
```

Figure 4.3 Ullmann’s algorithm
Although [Ullmann 1976] deals with only simple graphs, the condition above has been generalized to contain multiple edges and graph loops. The Ullmann’s algorithm is outlined in Figure 4.3.

4.1.2 The VF2 Algorithm

A graph and subgraph isomorphism algorithm is presented in [Cordella et al. 2001], which is known as VF2. Similarly to Ullmann’s subgraph isomorphism algorithm, VF2 is general in the sense that it does not impose any constraints on the input graphs. We consider only the subgraph isomorphism part of VF2 in the sequel. A high-level description of VF2 is depicted in Figure 4.4.

MATCH (Mapping $M$)
1. if $M$ covers all the nodes in $G^LHSG$
2. then $M$ is a match, save it, if only one match had to be found, terminate
3. else
4. $P$ = COMPUTE_CANDIDATE_PAIRS($)
5. for each $(n, m) \in P$
6. do if FEASIBLE($M, n, m$) then
7. add $(n, m)$ to $M$
8. MATCH($M$)
9. remove $(n, m)$ from $M$
10. end if
11. end for
12. end if

Figure 4.4 The outline of the VF2 algorithm

In the procedure detailed above, the following notations are used: two node variables $n \in G^H, m \in G^LHSG$, the set of the candidate pairs $P \subseteq V^H \times V^LHSG$, and $M \subseteq V^H \times V^LHSG$, for the mapping between a subset of $V^H$ and $V^LHSG$.

For reasons that will become apparent later, we introduce a simpler version of the FEASIBLE procedure here. This is basically a consistency check for the mapping: if $m \in G^LHSG$ has at most the same number of outgoing and incoming edges to or from the nodes already in the mapping as the corresponding node $n \in G^H$, then the mapping $M$ is consistent. The notations $pred(G, x)$ and $succ(G, x)$ refer to the set of nodes having incoming/outgoing edges to/from the node $x$ in graph $G$. The multiplicity of an $x, y$ element of a multiset $MS$ is denoted with $\mu_{MS}(x, y)$, and in Formulae (4.1.2) and (4.1.3), it corresponds to the number of edges between two vertices.

\[
m' \in M^LHSG \land \exists n'(n', m') \in M, n' \in pred(G^H, n) \land \mu^{LHSG}_{\text{in}}(m', m) \leq \mu^{H}(n', n) \quad (4.1.2)
\]

\[
m' \in M^LHSG \land \exists n'(n', m') \in M, n' \in succ(G^H, n) \land \mu^{LHSG}_{\text{out}}(m, m') \leq \mu^{H}(n, n') \quad (4.1.3)
\]

The validation is performed by the SIMPLE_FEASIBLE procedure (Figure 4.12).
SIMPLE_FEASIBLE (Mapping $M$, Node $n$, Node $m$)

1. for each predecessor $m'$ of $m$
2.   do if not Eq. (4.1.2)
3.     return false
4.   end if
5. end for
6. for each successor $m'$ of $m$
7.   do if not Eq. (4.1.3)
8.     return false
9.   end if
10. end for
11. return true

Figure 4.5 A simple Feasible procedure

The validation checked by the procedure SIMPLE_FEASIBLE is necessary if one wants to have correct matches. There are, however, optional conditions which can accelerate the algorithm in case of the vast majority of the input graphs, but a correct and consistent match can be achieved without them. The VF2 algorithm enforces two more types of such constraints described in [Cordella et al. 1999]. These heuristics are illustrated by the FEASIBLE procedure. The terminal sets (4.1.4)-(4.1.7) contain the nodes adjacent to the ones already included in the match, but themselves are not included in the match. Then we check whether the number of the nodes adjacent to the new candidate pair $(n,m)$ is consistent in both graphs. The notation of the sets is parameterized by the incoming/outgoing property of the edges, and the graph which the nodes in the set belongs to. The notation $M^H$ and $M^{LHS}$ are the projections of $M$ onto $V^H$ and $V^{LHS}$, respectively.

$$T^H_{in} = \{x \mid x \in \text{pred}(G^H, y), y \in M^H, x \notin M^H, x, y \in G^H\} \quad (4.1.4)$$
$$T^{LHS}_{in} = \{x \mid x \in \text{pred}(G^{LHS}, y), y \in M^{LHS}, x \notin M^{LHS}, x, y \in G^{LHS}\} \quad (4.1.5)$$
$$T^H_{out} = \{x \mid x \in \text{succ}(G^H, y), y \in M^H, x \notin M^H, x, y \in G^H\} \quad (4.4.6)$$
$$T^{LHS}_{out} = \{x \mid x \in \text{succ}(G^{LHS}, y), y \in M^{LHS}, x \notin M^{LHS}, x, y \in G^{LHS}\} \quad (4.4.7)$$

Then the first type of the conditions can be expressed in the following forms:

$$\#\{\text{pred}(G^{LHS}, m) \cap T^{LHS}_{in}\} \leq \#\{\text{pred}(G^H, n) \cap T^H_{in}\} \quad (4.4.8)$$
$$\#\{\text{succ}(G^{LHS}, m) \cap T^{LHS}_{in}\} \leq \#\{\text{succ}(G^H, n) \cap T^H_{in}\} \quad (4.4.9)$$
$$\#\{\text{pred}(G^{LHS}, m) \cap T^{LHS}_{out}\} \leq \#\{\text{pred}(G^H, n) \cap T^H_{out}\} \quad (4.4.10)$$
$$\#\{\text{succ}(G^{LHS}, m) \cap T^{LHS}_{out}\} \leq \#\{\text{succ}(G^H, n) \cap T^H_{out}\} \quad (4.4.11)$$

The second type of constraints examines the nodes either in $M$ or the sets (4.1.4)-(4.1.7) in both input graphs.

$$\#\{x \mid x \in \text{pred}(G^{LHS}, m), x \notin M^{LHS}, (w, x) \notin (T^H_{out} \cup T^{LHS}_{in})\} \leq$$
$$\#\{y \mid y \in \text{pred}(G^H, n), y \notin M^H, (y, z) \notin (T^{LHS}_{out} \cup T^{LHS}_{in})\} \quad (4.1.12)$$
$$w \in G^H, z \in G^{LHS}$$
The validation suggested by VF2 is realized by the FEASIBLE procedure (Figure 4.6).

FEASIBLE (Mapping $M$, Node $n$, Node $m$)
1 if not SIMPLE_FEASIBLE (Mapping $M$, Node $n$, Node $m$) return false
2 if not Eq. (4.1.8)- Eq. (4.1.11) then return false
3 return true

Figure 4.6 The feasibility check of the VF2 algorithm

In the general MATCH algorithm (Figure 4.4), Line 4 computes the set $P$ of the pairs which are candidates for inclusion in $M$. The procedure for this computation is shown in Figure 4.7.

COMPUTE_CANDIDATE_PAIRS()
1 if $\emptyset \neq T^\text{out}_H \cap T^\text{out}_\text{LHS}$ then $P = T^\text{out}_H \times \{\min T^\text{out}_\text{LHS}\}$
2 else if $\emptyset \neq T^\text{in}_H \cap T^\text{in}_\text{LHS}$ then $P = T^\text{in}_H \times \{\min T^\text{in}_\text{LHS}\}$
3 else $P = (V^H - M^H) \times \{\min(V^\text{LHS} - M^\text{LHS})\}$

Figure 4.7 Computing the candidates in the VF2 algorithm

4.2 Properties of instantiation

In this section, we examine the valid instantiations related to the multiplicity values, with respect to the case where multiplicity values consist of a nonzero integer element.

Proposition 4.2.1
The class diagram depicted in Figure 4.8 can be instantiated by $na'$ objects of type A and $nb'$ objects of type B, where $n$ is an arbitrary positive integer and

\[ a' = \frac{a}{\text{GCD}(a, b)} \]  
\[ b' = \frac{b}{\text{GCD}(a, b)} \]

where GCD denotes the greatest common divisor.

Figure 4.8 A general bidirectional association with nonzero integer multiplicities

Proof: It follows from the UML class diagram semantics that each object of type $A$ must have exactly $b$ number of $B$ type objects. Assume we have $y$ number of $A$ type
objects. Then $by = z$ edges exist on the object level. Similarly, if $x$ number of $B$ type objects exist, $ax = z$ edges are set up, and we solve the Diophantine equation (LCM denotes the least common multiple):

$$ax = by = n\text{LCM}(a, b)$$

$$ax = n\text{LCM}(a, b)$$

$$x = \frac{n\text{LCM}(a, b)}{a}$$

Because of the symmetry of the model (Figure 5.1):

$$y = \frac{n\text{LCM}(a, b)}{b}$$

Using a well-known formula we can express (4.2.5) and (4.2.6) as a function of the other multiplicity variable:

$$\text{GCD}(a, b)\text{LCM}(a, b) = ab$$

$$\text{LCM}(a, b) = \frac{ab}{\text{GCD}(a, b)}$$

$$x = \frac{nb}{\text{GCD}(a, b)}$$

$$y = \frac{na}{\text{GCD}(a, b)}$$

Thus we have proven the proposition.

In the previous proof, we have solved a linear Diophantine equation system. Because of the simplicity, we could give the solution in closed form; thus, we do not need the widely used solution mechanisms, which are also applicable to more complex cases.

Based on Proposition 4.2.1 an example can be created (Figure 4.9):

![Figure 4.9 Parallel links](image)

Although this is a construct that conforms to the UML standard [OMG UML 2.0], multiple edges are not used in practical applications. If multiple edges are not allowed, as it is the case regarding the LHS rules, another proposition can be used.

**Proposition 4.2.2**

If no multiple edges are allowed for associations in the object diagram, the class diagram in Figure 5.1 can be instantiated by $na$ objects of type A and $nb$ objects of type B, where $n$ is an arbitrary positive integer.

**Proof.** We use the $x, y$ variables from the previous proof. As a first step an $A$ node is taken and tied to $b$ number of different $B$ type nodes to avoid multiple edges. It
means that this step requires $b$ number of $B$ type nodes to be available. We do not restrict the method for selecting the connection during a particular step, but it always holds that in worst case the step $(ak+l)$ requires the availability of $(k+l)b$ $B$ type objects $(k=1,2,...,N)$. This is because the worst case does not use a new $B$ type object until it is necessary: when all the already tied $B$ type objects are exhausted. Consequently:

$$a(k-1)+1 \leq s \leq ak \Rightarrow kb \leq x,$$  \hspace{1cm} (4.2.11)

where $s$ denotes the ordinal number of the current step. Each step uses and exhaust the connection facilities of exactly one $A$ type object, hence $s = y$.

$$a(k-1)+1 \leq y \leq ak \Rightarrow kb \leq x$$  \hspace{1cm} (4.2.12)

Recall that because

$$ax = by$$  \hspace{1cm} (4.2.13)

holds for the edge numbers

$$a(k-1)+1 \leq y \leq ak \Rightarrow kb \leq \frac{by}{a}$$  \hspace{1cm} (4.2.14)

$$a(k-1)+1 \leq y \leq ak \Rightarrow ak \leq y,$$  \hspace{1cm} (4.2.15)

Then the only solution:

$$ak = y,$$  \hspace{1cm} (4.2.16)

and the symmetry suggests

$$bk = x.$$  \hspace{1cm} (4.2.17)

As $k=0$ is not considered as a solution for semantic reasons, Proposition 4.2.2 directly follows from Formulae (4.2.16) and (4.2.17).

Because this is a special case of the multiple edge version, if we do not simplify (4.2.1) and (4.2.2) with the greatest common divisor, we obtain Eq. (4.2.1) and Eq. (4.2.2) given in Proposition 4.2.1.

### 4.3 Instantiating the model structure

It is worth examining which $n$ values should be checked for a specific $a$ value. It would be advantageous if we could formulate an upper bound for $n$ to be examined, and decide whether it can be a part of the match. Unfortunately, in general such an upper bound cannot be given in all cases; further parts of the object graph have to be examined.

Assume that $ma$ number of $A$ type objects and $mb$ number of $B$ type objects are available. Examining $na$ and $nb$ objects, it cannot be decided whether these objects form a valid instantiation if $n < m$. We create an instance construct with a parameter $n$, as it is depicted in Fig 4.10. The structure contains $m$ number of blocks and each block contains $a$ number of $A$ type objects and $b$ number of $B$ type objects. As it follows from Proposition 4.2.2, a block can form a satisfying instantiation of the given class diagram. However, if an edge is removed from the blocks, which are instances in themselves, each block will contain an $A$ and a $B$ type objects, respectively, that are able to accept an edge.
This edge is drawn from a free $A$ type object of the $n^{th}$ block to a free $B$ type object of the $(n+1)^{th}$ block. In the case of the $m^{th}$ block, the first block is considered. Consequently, the threshold for examined $n$ values should be

$$\min \left( \frac{\#A}{a}, \frac{\#B}{b} \right)$$ \hspace{1cm} (4.3.1)$$

If we want to use UML class diagram formalism for describing patterns, we have to examine the instantiation process on a mathematical basis. For instance there are patterns for which no valid instantiation exists. In order to deal with this problem, we need to determine the number of objects participating in a valid instantiation and we need to discover how it depends on the multiplicity values.

For computing the allowed numbers of the participating objects in the whole instantiation model graph, we consider a specific representation derived from the metamodel, which is referred to as Incidence Matrix with Multiplicity (IMM) in the sequel [Levendovszky et al. 2005c] [Levendovszky & Charaf 2005]. The matrix is given by its creation algorithm (Figure 4.11).

CREATEIMM( )
1 Traverse the model graph
2 Take each edge $e_j$.
3 If vertices $v_k$ and $v_l$ are the incident upon $e_j$, then set $IMM_{kj}$ to the $v_k$ side multiplicity and $IMM_{lj}$ to the $v_l$ side multiplicity of $e_j$.

Figure 4.10 An infinite construct for a class diagram which cannot be analyzed by blocks

Figure 4.11 The CreateIMM algorithm
IMM can be considered as a short representation of the equations established for each node based on Proposition 4.2. In our example (Figure 4.12) the node equations can be written in the following equation system:

\[
\begin{align*}
A : x_0 &= x_1 \\
B : 3x_0 &= x_1 \\
C : 2x_1 &= 6x_2 \\
D : x_2 &= x_3
\end{align*}
\]

(4.3.2)

Obviously, in the formula above, all \( x_i \) variables are nonzero integers.

\[
\begin{bmatrix}
1 & 0 & 0 & 1 \\
3 & 1 & 0 & 0 \\
0 & 2 & 6 & 0 \\
0 & 0 & 1 & 1
\end{bmatrix}
\]

Figure 4.12 An example for creating IMM.

To solve these sorts of equations, an elimination algorithm is proposed (Figure 4.13). The source code of the algorithm can be downloaded from [VMTS].

```
ELIMINATION(IMM imm)
1 for each j column index
2    r0=index of row with first nonzero element
3    r1=index of row with second nonzero element
4    if there is no r0 and r1 then break
5    lcm=LCM(imm[r0,j],imm[r1,j])
6    MultiplyRow(r0, lcm/imm[r0,j])
7    MultiplyRow(r1, lcm/imm[r1,j])
8 for each j2 column index
9    if imm[r0,j2]= imm[r1,j2] then
10       imm[r1,j2]=0
11    else if imm[r0,j2]!=0 and imm[r1,j2]!=0 then
12       Inconsistent parallel paths. No instantiation.
13    else if imm[r1,j2]!=0
14       imm[r0,j2]=imm[r1,j2];
15       imm[r1,j2]=0;
16    end if
17 end for
18 end for
19 rowlcm=LCM of the 0th row of imm
20 for each j column index
21    imm[0,j]=rowlcm/ imm[0,j]
22 end for
```

Figure 4.13 The IMM Elimination algorithm
After the elimination, the 0th row of the IMM contains a factor $f_i$ for each edge $e_i$. Each $m_1, m_2$ multiplicity on an edge $e_i$ must be multiplied by the factor $f_i$. The number of the $m_i$ side node can be $k_{fm_i}^1$, where $k$ is an arbitrary nonzero integer and the cardinality for the rest of the nodes can be computed similarly. An example for the elimination algorithm:

$$
\begin{bmatrix}
1 & 0 & 0 & 1 \\
3 & 1 & 0 & 0 \\
0 & 2 & 6 & 0 \\
0 & 0 & 1 & 1 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
3 & 0 & 0 & 3 \\
3 & 1 & 0 & 0 \\
0 & 2 & 6 & 0 \\
0 & 0 & 1 & 1 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
6 & 2 & 6 & 6 \\
0 & 0 & 0 & 0 \\
0 & 2 & 6 & 0 \\
0 & 0 & 1 & 1 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
1 & 3 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\Rightarrow (\text{LCM}=6),
$$

$$
x_0 = 1 \quad A: 1n
$$

$$
x_1 = 3 \quad B: 3n
$$

$$
x_2 = 1 \quad C: 6n
$$

$$
x_3 = 1 \quad D: 1n
$$

As a second example we consider an example depicted in Figure 4.14.

![Figure 4.14 An UML class diagram having no valid instance](image)

Applying the IMM algorithm to the counterexample:

$$
\begin{bmatrix}
2 & 1 \\
3 & 1 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
6 & 3 \\
6 & 2 \\
\end{bmatrix}
\Rightarrow \text{Inconsistent equation.}
$$

The following example also leads to an inconsistent equation. Using this tool, we show that a systematic tool can recommend a correct, consistent multiplicity set in special cases.

$$
\begin{bmatrix}
3 & 0 & 0 & 0 & 7 \\
2 & 5 & 1 & 0 & 0 \\
0 & 4 & 0 & 1 & 0 \\
0 & 0 & 0 & 5 & 1 \\
0 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
24 & 60 & 12 & 15 & 56 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 15 & 3 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\Rightarrow \text{Inconsistent equation.}
$$

Although this equation cannot be solved, one can offer a solution for the multiplicities in the last column:

$$
8m_1 = 3m_2 = \text{lcm}(8,3) = 24, l = 1, 2, \ldots
$$

$$
m_1 = 3
$$

$$
m_2 = 8
$$

Substituting the result into the original matrix:
It can be seen that the rows of the IMM matrix represent the equations to be solved. In order to prove that the IMM algorithm provides all the correct solutions to the problem, we have to examine how the steps of the algorithm influence the equation system.

**Proposition 4.3**

The matrix resulted by an arbitrary elimination step represents an equation set which is equivalent to the multiplicity equations established according to the instantiation equation. Furthermore, there is no solution which can form a valid instantiation, and it is not among the results of the IMM algorithm.

**Proof.** The IMM constructed by the `CreateIMM` procedure is a representation of the equation established according to the instantiation equation, such that each row of the IMM contains an equation. Because the distinction between the set of equations and the IMM lies in the representation only, this is equivalent to the original set of equations. Line 6 and 7 of the elimination algorithm multiplies two equations by an integer, which results in an equivalent set of equations. Lines 8-17 merge two equations

\[
ax_1 = a_2 x_2 = \ldots = a_n x_n \quad (4.3.8)
\]

\[
ax_1 = b_2 x_2 = \ldots = b_n x_n \quad (4.3.9)
\]

into one equation:

\[
ax_1 = c_2 x_2 = \ldots = c_n x_n \quad (4.3.10)
\]

This merging step can be accomplished if there is no contradiction between the individual coefficients. The value zero means that the variable is not concerned, so zero can always be replaced with a coefficient residing in another row during the merging process. But if there are two coefficients different from zero at the same position in the two matrix rows, merging cannot be completed because of contradiction. If merging can be performed, the new equation remains equivalent to the unmerged ones, which follows from the transitive property of the equality. Lines 20-21 replace the coefficients with the solution based on (4.3.2).

\[
d_1 x_1 = d_2 x_2 = \ldots = d_n x_n = nLCM(d_1, d_2, \ldots, d_n) \quad (4.3.11)
\]

To prove the second part of the proposition, we assume that there is a solution vector \(X\) which forms a valid instantiation, but it is not a part of the solution provided by the IMM elimination algorithm. If \(X\) can occur in a valid instantiation, it has to satisfy the equations appearing in the instantiation equation. As it has been proven above, the set of equations remain equivalent during the elimination process; thus, it must also be the solution of the equation set resulted by the elimination process.
4.4 Type-aware matching with heuristics

Section 4.1 summarized two algorithms applying different graph representations. In this section, both methods are extended with metamodel-related data structures and heuristics. These methods often perform better in time than their original counterparts, but their storage space requirements are higher, because of the additional metainformation.

First of all, the VF2 algorithm must be capable of checking the type information during the matching process. Since this is a consistency check, it is implemented in the SIMPLE_FEASIBLE procedure (Figure 4.15).

```
1 if not typeof(n)=typeof(m) return false;
2 for each predecessor m’ of m
3     if not (4.4.1)
4     return false
5 end if
6 end for
7 for each successor m’ of m
8     if not (4.4.2)
9     return false
10 end if
11 end for
12 return true
```

Figure 4.15 A simple Feasible procedure

The conditions have to check the edge types as well:

\[
\forall (\text{typeof}(n,n’)) \mu_{LHS}^{\text{pred}} (m’,m) \leq \mu_{LHS}^{\text{pred}} (n’,n) \land \text{typeof}(m’,m) = \text{typeof}(n’n) \\
\forall (\text{typeof}(n,n’)) \mu_{LHS}^{\text{succ}} (m’,m) \leq \mu_{LHS}^{\text{succ}} (n’,n) \land \text{typeof}(m’,m) = \text{typeof}(n’n)
\] (4.4.1)

The conditions (4.1.8)-(4.1.13) must be extended in a very similar way. Now we have a variant of the VF2 algorithm which is capable of working with metamodels, and is referred to as MetaVF2 in the sequel. Then, it is possible to add modifications that benefit from the availability of the metamodel and the metamodel-model mapping.

The first step of computing the candidate pairs (when all the terminal sets are empty, since the mapping itself is empty) is to match a node from the LHS graph to all the nodes in the host graph. The type equivalence of these nodes is checked by calling SIMPLE_FEASIBLE. We can reduce the cardinality of this set if after selecting an element from LHS, we walk down to the metamodel element representing its type and retrieve the nodes of this type that reside in the host graph.

**Proposition 4.4**

The worst case execution time of the COMPUTE_CANDIDATE_PAIRS() procedure (Figure 4.16) is the one provided by the VF2 algorithm (Figure 4.4), assuming a connected pattern graph.
**THE ALGORITHMIC BACKGROUND**

```plaintext
COMPUTE_CANDIDATE_PAIRS()
1 if $T^\text{out}_H \neq \emptyset \land T^\text{out}_{\text{LHS}} \neq \emptyset$ then $P = T^\text{out}_H \times \{\min T^\text{out}_{\text{LHS}}\}$
2 else if $T^\text{in}_H \neq \emptyset \land T^\text{in}_{\text{LHS}} \neq \emptyset$ then $P = T^\text{in}_H \times \{\min T^\text{in}_{\text{LHS}}\}$
3 else $P = \text{Instanceof}(\min(V^\text{LHS} - M^\text{LHS})) \times \{\min(V^\text{LHS} - M^\text{LHS})\}$
```

**Figure 4.16 Computing candidates in MetaVF2 algorithm**

**Proof.** The only difference between the MetaVF2 and this modified version is that in the first step (or in the first step of processing a new connected block of a disconnected pattern graph), the modified algorithm does not compute a Cartesian product of the given minimal LHS element and all host graph vertices

$$P = (V^H - M^H) \times \{\min(V^\text{LHS} - M^\text{LHS})\}, \quad (4.4.3)$$

but only those of them that are instances of the minimal LHS element

$$P = \text{Instanceof}(\min(V^\text{LHS} - M^\text{LHS})) \times \{\min(V^\text{LHS} - M^\text{LHS})\}. \quad (4.4.4)$$

In our model it is assumed that the $\text{Instanceof}$ operation takes $O(1)$ time complexity; thus, retrieving $(V^H - M^H)$ takes the same time as computing

$$\text{Instanceof}(\min(V^\text{LHS} - M^\text{LHS})). \quad (4.4.5)$$

This shortcut, however, is only beneficial when

$$\#(V^H - M^H) > \#\text{Instanceof}(\min(V^\text{LHS} - M^\text{LHS})). \quad (4.4.6)$$

If the pattern graph is connected, Line 3 in Fig. 4.16 can only be executed for the first time of the function call. Hence, for connected pattern graphs, it holds that

$$M^H = \emptyset, M^\text{LHS} = \emptyset. \quad (4.4.7)$$

Then

$$\#V^H < \#\text{Instanceof}(\min(V^\text{LHS})) \quad (4.4.8)$$

is impossible,

$$\#V^H = \#\text{Instanceof}(\min(V^\text{LHS})) \quad (4.4.9)$$

occurs only when every instance graph element has the same metaelement. Therefore, this is the worst case of the algorithm (same as the MetaVF2), for all other graph

$$\#V^H > \#\text{Instanceof}(\min(V^\text{LHS})). \quad (4.4.10)$$

Another offline heuristics assumes some simple statistical information about the model-metamodel relationship. If the first selected node is the instance of the metanode which has the fewest instances in the host graph, we minimize the

$$\#\text{Instanceof}(\min(V^\text{LHS})) \quad (4.4.11)$$

expression by choosing the $V^\text{LHS}$ vertex having the fewest instance. Unfortunately, this heuristics is not appropriate in every case. By choosing the $V^\text{LHS}$ vertex having the fewest instance in every step, it is possible that more steps are necessary than in case of the algorithm developed in Proposition 4.4. In order to contradict, we assume that by choosing the $V^\text{LHS}$ vertex having the fewest instances, less or the same number of steps are necessary than in case of the algorithm developed in Proposition 4.4. A simple counterexample can be seen in Figure 4.17, where starting with the fewest instances we need two steps to realize that no match exists, but only one step is enough if we start with a :B node.
We assume that during the model building, a vertex list is maintained in the metamodel ordered by the number of instances belonging to a vertex. Gathering the statistical information might be performed online (in matching time), but proving its advantages require a verification by simulation, which is the subject of future work.

4.5 A Metamodel-Based Matching Algorithm

This section is devoted to discussing the algorithmic background of the metamodel-based pattern matching. In Figure 4.18 two alternatives are depicted as valid matches for a pattern containing multiplicities more than one on both sides of the association. This inherent non-determinism follows from the UML standard itself, where both cases are a valid instantiation of the class diagram.

Our experience has shown that it is enough to examine only one block (the case where \( n=1 \) in the previous section) for practical applications. Thus, the algorithm provided in this section considers that case only.

In order to get closer to metamodel-based matching, we analyze the VF2 algorithm. It can be observed that each recursive level deals with an LHS element: it attempts to match all the possible pairs, including the LHS element in a particular matching situation. For a certain level, the `COMPUTECANDIDATEPAIRS()` function generates the pairs to be tested. The main difference between the metamodel-based and the basic case is that metamodel elements specify more than one input model node to be matched, and this number can be dynamic (i.e. depending on the host graph topology). This issue is resolved by placing virtual nodes in the candidate list in accordance with the results of the previous section. We define a list referred to as precedence list for the nodes connected to the current pattern element. The precedence list consists of nodes and an integer value assigned to each node, which denotes the multiplicity. Each node is placed in the list with the maximum multiplicity. If the match algorithm realizes that there is no more element to be added
to the match, the IS_FULL_MATCH function checks whether the current assignment is a valid match. The pseudo code of the full algorithm is listed in Figure 4.19.

```c
bool UMLFEASIBLE (Match M, HostGraphNode n, PatternNode m) {
    foreach matchPair in M {
        foreach hostNode in matchPair {
            if not CHECKTWOPOINT(hostNode, matchPair.LHSNode, n, m) then return false
        }
    }
    foreach lhsNeighbourNode in GETLHSNEIGHBOURNODES (matchPair.LHSNode) {
        if ISEDGEBETWEENRULENODES(lhsNeighbourNode, m) then
            bHasProperNode = false
        foreach hostNeighbourNode in GETHOSTNEIGHBOURNODES (matchPair.HostNode) {
            if HASPROPEREDGEBETWEENNODES(hostNeighbourNode, lhsNeighbourNode, n, m) then
                bHasProperNode = true
        }
        if not bHasProperNode then return false
    }
    foreach lhsNeighbourNode in GETLHSNEIGHBOURNODES (m) {
```
if ISEDGEBETWEENRULENODES(lhsNeighbourNode, m) then
bHasProperNode = false
foreach hostNeighbourNode in GETHOSTNEIGHBOURNODES(n)
if HASPROPEREDGEBETWEENNODES(hostNeighbourNode, lhsNeighbourNode, n, m) then
bHasProperNode = true
end foreach
if not bHasProperNode return false
end if
end foreach
return true

UML COMPUTE CANDIDATEPAIRS()
if $T_{H}^{out} \neq \emptyset \land PL_{H}^{out} \neq \emptyset$ then $P = T_{H}^{out} \times \{\min PL_{H}^{out}\}$
else if $T_{H}^{in} \neq \emptyset \land PL_{H}^{in} \neq \emptyset$ then $P = T_{H}^{in} \times \{\min PL_{H}^{in}\}$
else if $T_{H}^{out} \neq \emptyset \land T_{LHS}^{out} \neq \emptyset$ then $P = T_{H}^{out} \times \{\min T_{LHS}^{out}\}$
else if $T_{H}^{in} \neq \emptyset \land T_{LHS}^{in} \neq \emptyset$ then $P = T_{H}^{in} \times \{\min T_{LHS}^{in}\}$
else $P = \text{Ins} \tan \text{coef} \left(\min(V^{LHS} - M^{LHS})\right) \times \{\min(V^{LHS} - M^{LHS})\}$
end if

Figure 4.19 An UML-based pattern matching algorithm

The final algorithm has another optimization step compared to the original VF2 philosophy: only the connected elements are considered among the neighbor nodes when the feasibility of the algorithm is investigated.

The matching algorithm and the role of the precedence list in this process are illustrated via an example. In Figure 4.20 a metamodel and the related instance model are depicted. With the help of this construct, we present how the matching algorithm parses this instance model, assuming that the LHS graph of the rewriting rule is the same as the metamodel.

(a) Metamodel:

(b) Instance Model:

Figure 4.20 (a) Metamodel, (b) Instance Model of the Precedence List example

The transformation contains the rewriting rules, the ID of the model to which the transformation has to be applied, and a list with the pivot nodes which select the nodes in the LHS graphs, where the algorithm has to start the pattern matching. These parameters are passed to the matching algorithm. In the present case, the transformation contains only one rewriting rule and it is assumed that there is no passed pivot node; therefore, the algorithm selects the pivot node automatically based on the statistical information. The node $A$ will be the pivot node, because it has the fewest instances.
In the first step, the algorithm selects the instance nodes with type $A$ ($A1$), it adds the $A - A1$ pair to the match, and calls the matching method recursively.

In the second step, the algorithm searches for suitable $B$ type nodes in the instance model, based on the LHS graph; therefore, it selects the neighbor nodes of the $A1$ node with $B$ type: $B1$, $B2$ and $B3$. This means that the algorithm has three possible pairs: $B - B1$, $B - B2$ and $B - B3$. The algorithm selects, for instance, the $B - B1$ pair and validates the following conditions for this possible pair: checks the actual pair (i) against the actual match, (ii) against the nodes which are adjacent to the match and (iii) against the nodes which are adjacent to the actual pair ($B - B1$). The actual pair is feasible; hence, the algorithm checks the multiplicity, which is 3. Thus, it does not add the actual pair to the match, but creates a precedence item and adds it to the precedence list. The precedence item in the current case contains the following information: the LHS node which is already in the match ($A$), the actual LHS node ($B$), the edge between them (the edge in metamodel between the $A$ and $B$ nodes), the already checked (feasible) instance nodes ($B1$), the possible instance nodes ($B2, B3$) and the number of the instance nodes. With these parameters, the algorithm has to find nodes to satisfy the expected multiplicity ($3 - 1 = 2$).

After the precedence item creation, the algorithm calls itself recursively. In this call, the algorithm notices that the precedence list is not empty, therefore, it tries to match the elements in the list: the algorithm finds the feasible instance node (e.g. $B2$) in the list of possible instance nodes and decreases the number of the searched instance nodes to 1.

<table>
<thead>
<tr>
<th>Actual LHS Node</th>
<th>Actual Match</th>
<th>Already Checked Nodes</th>
<th>Possible Nodes</th>
<th>Number of Necessary Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td>$A - A1$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td>$B - A1$</td>
<td>$B1$</td>
<td>$B2, B3$</td>
<td>2</td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td>$B - A1$</td>
<td>$B1, B2$</td>
<td>$B3$</td>
<td>1</td>
</tr>
<tr>
<td><strong>Step 4</strong></td>
<td>$B - A1$</td>
<td>$B1, B1, B2, B3$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Step 5</strong></td>
<td>$C - A1$</td>
<td>$C1$</td>
<td>$C2, C3, C4, C5, C6$</td>
<td>5</td>
</tr>
<tr>
<td><strong>Step 6</strong></td>
<td>$C - A1$</td>
<td>$C1, C2$</td>
<td>$C3, C4, C5, C6$</td>
<td>4</td>
</tr>
<tr>
<td><strong>Step 7</strong></td>
<td>$C - A1$</td>
<td>$C1, C2, C3$</td>
<td>$C4, C5, C6$</td>
<td>3</td>
</tr>
<tr>
<td><strong>Step 8</strong></td>
<td>$C - A1$</td>
<td>$C1, C2, C3, C4$</td>
<td>$C5, C6$</td>
<td>2</td>
</tr>
<tr>
<td><strong>Step 9</strong></td>
<td>$C - A1$</td>
<td>$C1, C2, C3, C4, C5$</td>
<td>$C6$</td>
<td>1</td>
</tr>
<tr>
<td><strong>Step 10</strong></td>
<td>$C - A1$</td>
<td>$C1, C2, C3, C4, C5, C6$</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

| Table 4.1 The matching process |

In the fourth step, the $B3$ instance node is the next feasible node and the algorithm decreases the number of the searched instance nodes again, which in this step becomes 0. Thus, the algorithm removes the precedence item from the precedence list and adds the following pair to the actual match: $B - B1, B2, B3$. 

---

**THE ALGORITHMIC BACKGROUND** 57
In the fifth step, the algorithm searches for suitable $C$ type nodes in the instance model: $C_1, C_2, C_3, C_4, C_5$ and $C_6$. The algorithm creates a precedence item again, processes the $C$ type instance nodes step by step, and, finally, it adds the last pair to the match: $C - C_1, C_2, C_3, C_4, C_5, C_6$. Table 4.1 presents the process of the matching, step by step.

### 4.6 Chapter Summary

An algorithmic background for metamodel-based model transformation has been contributed in this chapter. The matching process has been accelerated with type-aware heuristics, which uses the naturally available metamodel of the host graph.

Then, the IMM elimination algorithm was provided, which is suitable for predicting the possible number of objects participating in a valid instantiation of a specific class diagram. The correctness of the algorithm has also been proven. In order to find a match for LHS consisting of metamodel elements, a pattern matching algorithm has been provided, which is an enhanced version of the VF2 subgraph isomorphism algorithm. Although this method has been developed for metamodel-based rewriting rules; it can be used by modeling environments to check whether a valid instantiation exists for a specified class diagram, when the navigable properties are turned on.

If there is not only a single value given, but rather an interval or a series of intervals, the VMTS matching algorithm executes the IMM algorithm for each value of the intervals. When no upper limit is given, the valencies (degrees) of the host graph nodes are considered.

These algorithms take advantage of the availability of the metamodels in environments like GME, GReAT, and VMTS. In the proposed solution, the memory requirements are higher than that of conventional algorithms, but the metamodels are naturally available because of the technique used in the transformation.

Considering the performance issues, the Ullmann’s algorithm is very popular, but performance measurements [Foggia et al. 2001] have shown that better performance can be achieved using VF2. These evaluation measurements, however, have taken place on randomly generated graphs. The examination of UML and DSL models are the subject of future research. Further work includes devising new heuristics based on metamodel parameters and instantiation statistics.

Currently, the heuristics can be given explicitly by the designer of the transformation; thus, it is ensured that the algorithm runs faster in case of the special models. Performance comparisons for larger models are the subject of future work.

As far as novelty is concerned, no other algorithm exists for the metamodel-based model transformation rule matching, apart from the contributed one.
5. Metamodeling and Homomorphic Mappings

In graph grammars and graph rewriting systems, an LHS or an RHS is correct as far as it is a well-formed graph. With the introduction of metamodel-based rewriting rules that are applied to models typed over a metamodel, not all rules can be executed, or can have correct semantical meaning, which are valid elements bound to existing types. In this chapter we propose a solution to this problem. In general, the topological relationship between a metamodel making heavy use of inheritance and its instance models are not really straightforward. We introduce the concept of the homomorphic metamodel to overcome this problem, and we prove the equivalence between the homomorphic metamodel and the original one.

In Section 5.1, the mappings are discussed. Using those concepts, Sections 5.2 and 5.3 elaborate a topological rule validation mechanism, which helps to validate the transformation rules. In this section, associations with zero multiplicities are not subjects of the investigation. The closely related publications can be found in [Levendovszky & Charaf 2004] [Levendovszky et al. 2004c] [Levendovszky et al. 2005a] [Levendovszky et al. 2005d].

5.1 Mappings

In this section, the abstract algebraic definitions are summarized and sometimes interpreted, which are necessary to the propositions and proofs presented later in this work. We focus on the definition and properties of graph homomorphism, but we provide the formal definition of graph isomorphism used already in Chapter 4.

**Definition 5.1 (homomorphism):**
Homomorphism is a map between two groups $G$ and $H$, $\varphi: G \rightarrow H$, if the group operation is preserved, i.e. for all $g_1, g_2 \in G$

\[
(g_1 g_2) \varphi = (g_1) \varphi (g_2) \varphi
\]

and the identity is mapped to identity:

\[
\varphi(e_G) = e_H
\]

In category theory the homomorphism is often referred to as general morphism.

**Definition 5.2 (injection):**
Let $f$ be a function with the domain $A$ and range $B$. Then $f$ is an injection, if $f(x) = f(y)$, it implies $x = y$, $x, y \in A$. Injection is also called one-to-one (1:1) mapping.

**Definition 5.3 (surjection):**
Let $f$ be a function with the domain $A$ and range $B$. Then $f$ is a surjection, if for any $b \in B$ there exists an $a \in A$, for which $f(a) = b$. Surjection is also called onto mapping.

**Definition 5.4 (bijection):**
A mapping is a bijection if it is injection and surjection.

**Definition 5.5 (isomorphism):**
A homomorphic mapping is an isomorphism, if the mapping is a bijection.

Now these definitions are applied to graphs. For the sake of notation conventions and the seamless discussion, we give a definition for labeled directed graphs (LDG)
and graph homomorphism. These follow the conventions applied in [Rozenberg 1997].

**Definition 5.6** (Labeled directed graphs, LDG): Let $\Omega_V$ and $\Omega_E$ be two given alphabet for node and edge labels, respectively. Then the labeled directed graph is a six-tuple:

$$G = \{G_V, G_E, s^G, t^G, lv^G, le^G\} \quad (5.1.3)$$

where $G_V$ is the set of vertices, $G_E$ is the set of edges; $s^G$ and $t^G$ are the source and target functions ($s^G, t^G: G_E \rightarrow G_V$), which map an edge to its source and target vertices, respectively; and finally $lv^G: G_V \rightarrow \Omega_V$ and $le^G: G_E \rightarrow \Omega_E$, which assign a label to a vertex and an edge from the appropriate alphabet.

We can consider UML models as an LDG, where the node and edge information (type, attribute) are stored in the label. One may store this information as an XML document (a standard approach to this can be found in OMG’s XMI specification [OMG XMI 2003]). Regarding the labels of the vertices and the edges as an XML document can be a suitable implementation strategy for a UML class diagram. Now we define graph homomorphism.

**Definition 5.7** (graph homomorphism for LDGs): A graph homomorphism $f: G \rightarrow H$ is a pair of functions: $f = \{f_V: G_V \rightarrow H_V, f_E: G_E \rightarrow H_E\}$, which preserves the sources, targets, and labels, that is, it satisfies the following conditions:

$$f_V \circ t^G = t^H \circ f \quad (5.1.4)$$
$$f_V \circ s^G = s^H \circ f_E \quad (5.1.5)$$
$$lv^H \circ f_V = lv^G \quad (5.1.6)$$
$$le^H \circ f_E = le^G \quad (5.1.7)$$

As it is pointed out in [Hahn & MacGillivray 2002], graph homomorphism need not be surjective or injective.

**Definition 5.8** (graph isomorphism): A graph isomorphism is a bijection between the vertices of two graphs, $G$ and $H$, provided that if and only if two vertices are adjacent in $G$ then they are adjacent in $H$, and there is the same number of edges between the adjacent pairs of vertices.

### 5.2 The Homomorphic type of Mapping

First we introduce the notion of type preserving mapping. Let graph Meta be a UML class diagram. Let graph Instance be a UML object diagram instantiating Meta. Type Preserving Mappings (TPM) are mappings from Instance to Meta, such that every model element in Instance is mapped to its type element in Meta. This mapping unnecessarily utilizes all meta elements (if one considers abstract classes, for instance) in Meta. Before resolving this issue, we define Type Preserving Mapping (TPM) in a more formal way.
Definition 5.8 (Type Preserving Mapping)

Let $Meta$ be a labeled, directed graph (LDG) with labels conforming to the UML class diagram. Let another LDG $Instance$ be given with labels conforming to the UML object diagram. Furthermore, we assume that $Instance$ instantiates $Meta$ according to the instantiation rules enforced by UML. There are two functions

$$i_V : lv_{Meta}, lv_{Instance} \rightarrow \text{Boolean} \quad (5.2.1)$$

$$i_E : le_{Meta}, le_{Instance} \rightarrow \text{Boolean} \quad (5.2.2)$$

which return true if and only if the $Meta$ graph element having the label given as the first function argument is the type of the $Instance$ element having the label given as the second argument. A mapping between an $Instance$ and a $Meta$ LDG is called Type Preserving Mapping (TPM) if and only if the $Instance$ element is mapped to its type elements and to no other elements.

Practically, if we consider an XMI-like approach outlined above, $i_V$ and $i_E$ can be implemented as a simple comparison between the XML tag representing the type of the $Instance$ element, and the XML tag holding the name of the $Meta$ element. In other tools this relationship may be denoted by a reference or other linking methods peculiar to the applied programming environment. In the following statement we connect graph homomorphism and TPM.

Proposition 5.1

Let $Meta$ be an LDG with labels conforming to the UML class diagram. Let another LDG $Instance$ be given with labels conforming to the UML object diagram. Furthermore, we assume that $Instance$ instantiates $Meta$ according to the instantiation rules enforced by UML. If $Meta$ does not contain inheritance relationship (nor abstract classes, which would be semantically meaningless without inheritance), then the TPM between the $Instance$ and $Meta$ graph is a graph homomorphism.

Proof. First a TPM is created between $Instance$ and $Meta$. Because every vertex and edge has a type, every $Instance$ element participates in the mapping. Since there is no inheritance, every instance element is mapped to only one $Meta$ element. It means that this mapping is formed by a pair of functions.

Now it is shown that two adjacent vertices of the $Instance$ graph are mapped to two adjacent vertices in $Meta$. Suppose we have $A$ and $B$ vertices which are connected in $Instance$ via link $\downarrow L$, but their types (A and B, respectively) are not connected in $Meta$ with $L$ which is the association type for $\downarrow$ (Figure 5.1). It implies that $Instance$ has violated the rules of instantiation (defined by the UML standard), the $Instance$ is not the instance of the $Meta$. That contradicts the condition part of the proposition.

![Figure 5.1 Instantiation](image)

Proposition 5.2

Let $Meta$ be an LDG with labels conforming to the UML class diagram. Let another LDG $Instance$ be given with labels conforming to the UML object diagram. In addition, we assume that $Instance$ instantiates $Meta$ according to the instantiation rules enforced by UML. Let $A_n$ denote the set of the associations connected to the $n^{th}$ layer of a class hierarchy. Let $L_n$ represent the set of the links instantiating the
elements of \( A_n \). \( \text{Instance} \) can be mapped to \( \text{Meta} \) via graph homomorphism if and only if no \( :O \) objects are attached to link from more than one \( L_n \) set, where \( A_n \) belongs to the class hierarchy of \( C \), which is the type of \( :O \).

**Proof.** Firstly, it is shown that if the conditions stated in the proposition are satisfied, a TPM exists. The objects of the classes not participating in an inheritance hierarchy are mapped to their own types. The objects of the classes participating in an inheritance hierarchy are mapped to the class which has the association instantiated by the link attached to the given object. This mapping is a TPM.

Then it is demonstrated that the adjacent vertices of \( \text{Instance} \) are mapped to adjacent vertices of \( \text{Meta} \). For vertices not participating in the inheritance hierarchy, the statement holds based on Proposition 5.1. For the vertices participating in inheritance hierarchy, this property is enforced by the condition. Because of the type compatibility, an \( :X \) object can be assigned to its exact type \( X \) or any ancestors of \( X \), but only to one of them. Thus the type assignment is chosen such that the association of the link is attached to it preserves the adjacency.

Secondly, the inverse direction is shown. We assume that the mapping can be created, but links from more than one inheritance level are instantiated and assigned to an \( :X \) object. Hence, this should be assigned to \( X \) or one of its ancestor. We cannot choose either of them, because if we select a class based on an association of a specific level, the other will violate the adjacency in \( \text{Meta} \), which contradicts the original assumption.

The direct consequence of the propositions above is that in general, the UML object diagram cannot be mapped to UML class diagram by graph homomorphism. Hence, from a metamodel, we must construct a new metamodel, which considers the practical aspects and has the same expressivity as the original, but the instantiation relationship is a homomorphic TPM. To achieve this goal the equivalence of metamodels needs to be defined.

**Definition 5.9** (equivalence of metamodels)
Let \( \text{Meta}_1 \) and \( \text{Meta}_2 \) be two LDGs, with labels conforming to the UML class diagram. \( \text{Meta}_1 \) and \( \text{Meta}_2 \) are equivalent if and only if any LDG provided with labels conforming to the UML object diagram i.e. any instance of \( \text{Meta}_1 \) is also an instance of \( \text{Meta}_2 \), and vice versa. If only one direction is satisfied, namely, all instances of \( \text{Meta}_1 \) are the instance of \( \text{Meta}_2 \), then \( \text{Meta}_2 \) is compatible with \( \text{Meta}_1 \). Hence, equivalence can be defined as bidirectional compatibility. If at least one instance of \( \text{Meta}_1 \) - that instantiates each element in \( \text{Meta}_1 \) - is contained by at least one instance of \( \text{Meta}_2 \), then \( \text{Meta}_2 \) is partially compatible with \( \text{Meta}_1 \).

It is worth noting that contrary to the compatibility property, partial compatibility is a symmetric mapping. Then, an algorithm is given which creates an equivalent metamodel where the instantiation relationship is a homomorphic TPM. An example for this algorithm can be found in [VMTS].

**Algorithm outline 5.1** (creating homomorphic meta)
1. Walk through the inheritance hierarchy and copy all inherited data (attributes associations) from the ancestors to the derived class, while the value of the variable multiplicities are taken care of.
2. Remove inheritance.
3. Remove all abstract classes.

The algorithm uses a wide-spread technique to eliminate inheritance: it copies all the data and relations to the derived classes, but we must be careful with the association multiplicities, because its value must be distributed among the new nodes: their sum must match the old value. To handle this, we use variable names to denote multiplicity and we refer to them from another association. A detailed discussion of this algorithm can be found in [Levendovszky & Charaf 2004].

**Proposition 5.3**
The homomorphic metamodel and its generator metamodel are equivalent.

**Proof:** If an instance LDG instantiates the generator metamodel, it instantiates its homomorphic metamodel as well. The algorithm removes abstract classes only, which are not concerned with instantiation at all, so all the object types (classes) have been preserved with their attribute and other properties. Associations connected to certain classes as well as the inherited associations of the classes, remain unchanged after the transformation. Hence, all the links that instantiate the generator metamodel also instantiate the homomorphic metamodel. If an instance LDG instantiates the homomorphic metamodel, it also instantiates its generator metamodel. For objects and classes, the statement of the proposition can be shown similarly to the inverse direction. The only changes in the associations are that some of the direct associations become inherited by the transformation. This change has no influence on instantiation.

The multiplicity values produced by the algorithm were designed such that it conforms to equivalence.

**Proposition 5.4**
Let $Meta_1$ and $Meta_2$ be LDGs, with labels conforming to the UML class diagram, and we assume that they are assumed to be homomorphic metamodels. If $Meta_1$ is compatible with $Meta_2$, $Meta_1$ is a (not always connected) subgraph of $Meta_2$, not regarding the multiplicity labels. Moreover, the following formula is always true:

$$\text{Meta}_2 \subseteq \text{Meta}_1$$ (5.2.3)

where $\text{Meta}_1$ and $\text{Meta}_2$ are the sets of the allowed multiplicity for the $Meta_1$ and $Meta_2$ sides, respectively, at the same topological position.

In case of partial compatibility, where the zero multiplicity values are not allowed, it is enough to enforce the following conditions for each corresponding multiplicity pairs:

$$\text{Sup} \text{Meta}_1 \leq \text{Sup} \text{Meta}_2$$ (5.2.4)

$$\text{Meta}_1 \cap \text{Meta}_2 \neq \emptyset$$ (5.2.5)

where Sup is the supremum (the least upper bound) of the set which contains the allowed multiplicity values.

**Proof:**
If every instance of $Meta_1$ also instantiates $Meta_2$, it implies the following: if we construct an instance of $Meta_1$ which instantiates all classes in $Meta_1$, then it means that $Meta_2$ must contain all the classes from $Meta_1$. 
For partial compatibility, the conditions mean that (i) if there is an association in \( Meta1 \) there must be one with at least with the same supremum (that implies the containment) (ii) we ensure that there is an instance of \( Meta1 \) that instantiates every element of \( Meta1 \), and there is an instance of \( Meta2 \) which contains it. The latter condition is not necessary but rather useful. It is rather obvious to check the situations detailed above for the allowed multiplicity sets.

We apply the following notations listed in Table 5.1 from now on.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>Host graph</td>
</tr>
<tr>
<td>GM</td>
<td>Host graph metamodel</td>
</tr>
<tr>
<td>H</td>
<td>Output model</td>
</tr>
<tr>
<td>HM</td>
<td>Output metamodel</td>
</tr>
<tr>
<td>L</td>
<td>LHS graph</td>
</tr>
<tr>
<td>LM</td>
<td>LHS metamodel</td>
</tr>
<tr>
<td>R</td>
<td>RHS graph</td>
</tr>
<tr>
<td>RM</td>
<td>RHS metamodel</td>
</tr>
</tbody>
</table>

Table 5.1 Notations

It is important to note that these are metamodels of the matches, meaning the same instances cannot appear twice. Thus, one more indirection step is needed in case of the metamodel-based rules where a type appears more than once. Based on Proposition 5.4, the following algorithm can be established.

**Algorithm outline 5.2** (Detecting gluing time RH error)

1. If the partial compatibility does not hold for the \( LM \) and \( GM \) (it can be checked using Proposition 5.4), we can signal an error message or automatically adjust it emitting a warning message. In case of \( RM \) and \( HM \), if the compatibility holds, no additional error check is necessary.
2. If only the partial compatibility is satisfied, then there are two cases: (i) When \( RM \) allows a larger instance (with greater multiplicity), then it is considered an error which can be detected and reported. (ii) When \( RM \) allows a smaller instance it can be a part of the instance specified by \( HM \). In this case, the rewriting might cause an error, but seamless situations are also possible. This type of error can only be detected at gluing time, checking the created changes again \( HM \).
3. When \( HM \) is not partially compatible with \( RM \), an error message should be reported or an automatic adjustment can be performed (based on the conditions in Proposition 5.4), along with warning.

**5.3 The Homomorphic Instantiation**

It is interesting to examine the opposite direction, namely, how a homomorphic instantiation mapping can be established between the metamodel and the instance model. From now on we will focus on this question. Our assumptions are the following.

- There is no inheritance relationship in the metamodel. This condition is not restrictive. Section 5.2 contains an algorithm which eliminates the inheritance relationship from the metamodel without affecting the possible instances.
- We assume nonzero single multiplicity values on the association ends in the metamodel.

**Proposition 5.5**
The instantiation relationship in an arbitrary metamodel that conforms to the aforementioned conditions is not homomorphism.

*Proof.* The proof is given by a counterexample. In Figure 5.3, a metamodel and a possible instantiation are depicted. Assuming positive \( a \) and \( b \) parameters, the contradiction is clear: at the bottom of each block there are two \( \mathcal{A} \) and \( \mathcal{B} \) objects which are not adjacent, but the corresponding meta elements \((A, B)\) are connected.

Because of the negative result of Proposition 5.5, another approach should be taken. If no homomorphic instantiation relation can be established between the metamodel and its whole instance graph, the instance graph can be composed of subgraphs. A homomorphic mapping can be created between these subgraphs and the metamodel.

**Proposition 5.6**
Each \( I \) instance graph having a metamodel, satisfying the assumptions presented at the beginning of this subsection, can be decomposed into \( I_1, I_2, \ldots, I_K \) non-isomorphic subgraphs, such that the instantiation relationship between the \( M \) metagraph and the \( I \) graph is a graph homomorphism, where \( I \) is an instance of \( M \).

*Proof.* The proof is done constructively by providing an algorithm which performs the required decomposition.

**Algorithm.** Decomposition to isomorphic subgraphs.
1. \( N\text{Incl} = E(I) \)
2. for \( (k = 1, N\text{Incl} \neq \emptyset, k++) \)
3. \( e \in N\text{Incl}, N\text{Incl} = N\text{Incl} \setminus e \)
4. \( em \in E(M), em \in \text{Instancesof}(em) \)
5. Traverse the meta graph starting from \( em \), and for each elements of \( M \) choose only one element of \( I \) and add it to \( I_k \), such that the chosen element is the instance of the metaelement, and \( I_k \) is always isomorphic to a subgraph of \( M \), and the isomorphic mapping is the instantiation. If a used element is included in \( N\text{Incl} \), it should be removed from the set.
6. end for

The algorithm maintains a set for the instance graph edges which are not yet included in one of the \( I_k \) sets. The algorithm runs until all the instance graph edges have been included in at least one \( I_k \) sets, thus this is a full decomposition of \( I \) (if all the edges are included it implies all the nodes are included as well). No two \( I_k \) graphs can be isomorphic, since when creating a new \( I_k \) graph, a new, not used edge is taken from \( N\text{Incl} \). The instantiation mapping has been maintained in the subgraphs as well.
When traversing the metagraph, always the adjacent elements are taken. It means that if we maintain the isomorphism, the adjacent elements remain adjacent in both graphs. Now we prove that this is always possible. As the first step we take an edge \( e \) in \( I \), and find the type of it in \( M \), which was denoted by \( em \). Then we take an edge \( en \) in \( M \) that is adjacent to \( em \). Afterwards, an edge must be found in \( I \), which is the instance of \( en \), and adjacent to \( e \). If there is no such an edge, it contradicts the instantiation rules, which would mean that \( I \) is not a valid instance of \( M \).

After proving the correctness of the algorithm, it is worth examining its complexity.

**Proposition 5.7**
The worst case complexity of the decomposition algorithm is
\[
O(\#E(I)\#E(M)).
\] (5.3.1)

**Proof.** The for loop runs \( \#E(I)−\#E(M) + 1 \) times, since in the worst case each for loop takes only one new edge, except in the first run, because it has to take \( \#E(M)−1 \) new edges to create the isomorphic graph \( I_k \). The complexity of traversing the metagraph by the edges and establishing the new \( I_k \) is \( \#c\#E(M) \), where \( c \) is constant. Provided that \( \#E(I)>>\#E(M) \), the overall complexity can be calculated as follows:
\[
(\#E(I)−\#E(M) + 1)\ast\#c\#E(M) =
\]
\[
=\#c\#E(I)\#E(M) − \#c\#E(M)^{2} + \#c\#E(M) ≈
\]
\[
= O(\#E(I)\#E(M))
\] (5.3.2)

which is the exact complexity stated in the proposition.

With Propositions 5.6 and 5.7, the metamodel-model direction has also been discovered.

### 5.4 Chapter Summary

Section 5.2 developed an approach where the metamodels are transformed to offer homomorphic **typeof** relationships to the models. To ensure the interchangeability between the transformed and the original metamodel, the definitions of equivalence and compatibility have been introduced. It has been proven that the original and the transformed metamodels are equivalent on the instance level.

Based on these closer homomorphic relationships, topological properties have been proven. The other direction has been investigated as well. The **typeof** mapping can be used for offline topological validation of the rewriting rules against the metamodel of the input graph. The main drawback of the approach is the transformation that is needed to create the homomorphic relationship. This transformation, however, can be performed by a relatively simple algorithm. Considering novelty, no other approach is known to us which uses homomorphic mappings between the model and the metamodel or vice versa.
6. Parallel Rule Execution

In this chapter we discuss the parallel execution of metamodel-based rules. In order to use the theoretical background developed in the field so far, we have to build this problem on categorical constructs. Since we mainly consider the category Graph, the integration of this chapter is seamless to the previous ones. In Section 6.1, we introduce the necessary elements of category theory; in Section 6.2, the related work from the field of graph rewriting is provided, stressing the categorical algebraic approaches.

6.1 Category Theory

Category theory [Pierce 1991] [Barr & Wells 1999] [Segura 2001] [Sabetzadeh 2003] is a relatively young branch of algebraic topology, which can be considered a pure theory of functions instead of the traditional set-based approach. A natural question is why category theory is used to describe graph rewriting rules. As Hartmut Ehrig, the inventor of the DPO approach writes [Ehrig 1979]: “At least for the proofs, however, categorical techniques are highly desirable (e.g. the proof of the Church-Rosser Property I was reduced to 1/5 of the original length using pushout composition and decomposition techniques). On the other hand category theory is a language which allows speaking about complex structures in a short but precise way.”

Category theory focuses on functions; their domains and ranges are of less importance than they are in the set theoretical approach. Its basic constructs are morphisms (arrows), which lend themselves to a composition operator in an associative way. A category consists of arrows, their domains, and ranges that are called objects. Since category theory is not a widely used, well-known, formal background for computer scientists, a brief introduction is given here. The primary references are [Pierce 1991] [Barr & Wells 1999].

Definition 6.1 (Category):
A category C comprises:
1. Objects
2. Arrows (morphisms)
3. Operations assigning to each arrow f an object, an object dom f, its domain, and an object cod f, its codomain (denoted with f: A→B, if dom f: A and cod f: B). The collection of all arrows with domain A and codomain B is denoted with C(A,B).
4. A binary composition operator assigning to each pair of arrows f and g with cod f=dom g a composite arrow g o f: dom f→ cod g, satisfying the following associative law:
   for any arrows f: A→B, g: B→C and h: C→D (with A, B, C, D not necessarily distinct),
   \[ h \circ (g \circ f) = (h \circ g) \circ f \] (6.2.1)
5. For each object A there exists an (identity) arrow idA: A→A, satisfying the following identity law: for any arrow f: A→B,
   \[ idA \circ f = f, \text{ and } f \circ idA = f. \] (6.2.2)

One can create a directed graph, where the vertices are LDGs and the edges are homomorphic mapping between the LDG vertices. Thus we have obtained an
informal notion of the category of the directed graphs. To clarify the notations and the basic constructs, the well-known category of LDGs, called Graph, is defined.

**Definition 6.2 (Category of LDGs, Graph):**
1. Objects: LDGs,
2. Arrows: LDG homomorphisms,
3. Domain: the domain of the homomorphism; the codomain is the range of the homomorphism.
5. The identity arrows are the identity functions for both vertices and edges.

Then, it is obvious for an arbitrary \( f: A \rightarrow B \) that
\[
\text{id}_B \circ f = f \quad \text{and} \quad f \circ \text{id}_A = f.
\]

For the composites we need to show that they are homomorphisms as well. Suppose \( \phi: G \rightarrow H \) and \( \psi: H \rightarrow K \) mappings are graph homomorphisms, \( u \in G_E, \ m, n \in G_V, \) and \( s^G(u) = m, \ t^G(u) = n \). Based on the definition of the homomorphism:
\[
\phi_E(u) \in H_E \quad (6.2.4)
\]
\[
\phi_V(m), \phi_V(n) \in H_V \quad (6.2.5)
\]
\[
s^H(\phi_E(u)) = \phi_V(m) \quad (6.2.6)
\]
\[
t^H(\phi_E(u)) = \phi_V(m) \quad (6.2.7)
\]
Applying \( \psi \) homomorphism using the definition again:
\[
\psi_E(\phi_E(u)) \in K_E \quad (6.2.8)
\]
\[
\psi_V(\phi_V(m), \psi_V(\phi_V(n)) \in K_V \quad (6.2.9)
\]
\[
s^K(\psi_E(\phi_E(u))) = \psi_V(\phi_V(m)) \quad (6.2.10)
\]
\[
t^K(\psi_E(\phi_E(u))) = \psi(\phi_V(m)) \quad (6.2.11)
\]
It means that \( \psi \circ \phi \) is also a homomorphism. Graph homomorphisms can be thought of as creating composite functions, which are associative:
\[
h \circ (g \circ f) = (h \circ g) \circ f, \quad (6.2.12)
\]

Since the objects and arrows can easily be very complex, a graphical representation is used to make them more intuitive. Several category theoretical proofs can also be given “visually”.

**Definition 6.3 (Diagram):**
A diagram in a category \( C \) is a collection of vertices and directed edges, consistently labeled with objects and arrows of \( C \) (i.e. if an edge in the diagram is labeled with an arrow \( f \), and \( f \) has a domain \( A \) and a codomain \( B \), then the endpoint of this edge must be labeled with \( A \) and \( B \)).

The diagrams make the statements more comprehensible in category theory. The next step is to provide a mechanism to express equations with diagrams.

**Definition 6.4 (Commutative diagram):**
A diagram \( D \) in a category \( C \) is said to be commutative (or commutes) provided for any nodes \( i \) and \( j \) in the graph of \( D \) and any two paths
from $i$ to $j$ in the graph of $D$, the two paths

\[
Ds_n \circ D_{s_{n-1}} \circ \ldots \circ D_s = Dt_m \circ D_{t_{m-1}} \circ \ldots \circ Dt_1
\]

(6.2.15)

An illustrative example for commutative diagrams as equations is shown by describing the graph isomorphism in terms of the newly introduced categorical constructs.

The correctness of the categorical equation (4.6.16) can be checked based on Definition 5.7. Having constructed the definition of diagrams, the definition of a pushout is provided in two steps.

**Definition 6.5 (coproduct):**

The sum, also called the coproduct, $A+B$ of two objects in a category, consists of an object called $A+B$ together with arrows $\text{in}_1: A \rightarrow A+B$ and $\text{in}_2: B \rightarrow A+B$ such that given any arrows $f: A \rightarrow C$ and $g: B \rightarrow C$, there is a unique arrow $[f, g]: A+B \rightarrow C$ for which Figure 6.1 commutes.

![Figure 6.1 Coproduct (sum)](image)

In **Graph**, the coproduct is constructed by taking the disjoint union of the edge and the vertex set.

**Definition 6.6 (Disjoint Union)**

A disjoint union of two sets, $X$ and $Y$, is a binary operator that combines all distinct elements of a pair of given sets, while retaining the original set memberships as a distinguishing characteristics of the union set.

\[
X + Y = (X \times \{0\}) \cup (Y \times \{1\})
\]

(6.2.17)
Extending the coproduct with a commutative square, we obtain the notion of pushout (Figure 6.2).

**Definition 6.7** (pushout, pushout complement):
Let $C$ be a category, with two given arrows $b : A \rightarrow B, c : A \rightarrow C$. The \( \{D, g : B \rightarrow D, f : C \rightarrow D\} \) triple is called the pushout of \( \{b, c\} \) if the following conditions hold:
1. $g \circ b = f \circ c$ (commutative property)
2. For any object $D'$ and arrows $g' : B \rightarrow D'$, and $f' : C \rightarrow D'$ if $g \circ b = f \circ c$, there exists a unique arrow $h : D \rightarrow D'$ such that $h \circ g = g'$ and $h \circ f = f'$ (universal property).

![Figure 6.2 A diagram representation of pushout](image)

The triple $\langle C, c : A \rightarrow C, f : C \rightarrow D \rangle$ is called the pushout complement of $\langle b, g \rangle$, and $C$ is called $\langle b, g \rangle$ the pushout complement object of $\langle b, g \rangle$. $D$ is called a pushout object of $\langle b, c \rangle$.

As it can be seen, the commutative property is expressed by a square in the pushout diagram, in accordance with the definition of the commutative diagram.

After the general definitions, the category **Graph** is examined. To express the pushout in **Graph**, some set theoretical constructs are delineated.

**Definition 6.8** (equivalence relation, equivalence class, quotient set)
An equivalence relation $\sim$ on a set $X$ is a binary relation on $X$ that is reflexive ($a \sim a$), symmetric (if $a \sim b$, then $b \sim a$), and transitive (if $a \sim b$ and $b \sim c$, then $a \sim c$).

Given a set $X$ and an equivalence relation $\sim$ on $X$, the equivalence class of an element $a \in X$ is the subset of all elements in $X$ which are equivalent to $a$.

$$[a] = \{x \in X \mid x \sim a\} \quad (6.2.18)$$

The set of all equivalence classes in $X$ is called the quotient set of $X$ by $\sim$. It is denoted by $X/\sim$.

The most evident example for these constructs is the set of positive integers ($X = \mathbb{N}^+$), and the equivalence relation is the congruence modulo $m$. Then the quotient set is the set of the remainder classes.

As the second example, let $X$ be the set of bowling balls, and the equivalence relation be the color: if two balls are of the same color, they are considered equivalent, otherwise not. Then the quotient set is the set of ball colors.

The third example is about two graphs. Since graphs consist of two sets (edges and nodes), the operation can be accomplished componentwise.
Figure 6.3 shows three graphs, the quotient set of \( G_2 \) and \( G_3 \) should be computed. \( G_2 \) and \( G_3 \) are assumed to be disjoint. The equivalence relation (\( \approx \)), however, has been specified in a rather “weird” way: a third graph is given (the upper graph \( G_1 \) in Figure 6.3) and if two nodes, one chosen from \( G_2 \) and one from \( G_3 \), are mapped to the same node of \( G_1 \), they are in the same equivalence class. Thus the quotient set of the construct \( \left( (G_1 \cup G_2) / \approx \right) \) is depicted in Figure 6.4.

As it can be observed, \( G_2 \) and \( G_3 \) have been “glued together” based on a “blueprint” provided by \( G_1 \) and the mappings. The only problem is that if \( G_2 \) and \( G_3 \) are not disjoint, the union may “mix” the two graphs together, preventing the gluing from being performed according to the “blueprint”. Therefore, instead of the union, a disjoint union is applied.

Having provided a set theoretical description of gluing two graphs together, we connect this construct to its categorical description. Using the notations in the definition of pushout, given two morphism \( b: A \rightarrow B \), \( c: A \rightarrow C \), the pushout \( \langle D, g: B \rightarrow D, f: C \rightarrow D \rangle \) can be constructed as \( D = (B_v + C_v) \approx, D = (B_e + C_e) \approx \) i.e. the quotient set of the disjoint union modulo \( \approx \left\{ f(a), g(a) \mid a \in A \right\} \), smallest equivalence relation that maps each element to its equivalence class. Recall that in case of graphs, these operations must be performed on the vertex and the edge sets, respectively. The “blueprint” is provided by the graph object \( A \), the mappings \( f, g \) graph morphisms supply the mappings to the “blueprint”, and finally, the pushout object graph \( D \) is resulted by gluing together \( B \) and \( C \).

The existence of a unique pushout object is guaranteed by the universal property in the definition, but there is no such a guarantee for the pushout complement object. It can be proved that in case of the \textbf{Graph} category, the conditions are as follows.
• Dangling edge condition: There is no edge between $e \in D_E - g_E(B_E)$: it is incident upon any node in $g_v(B_v - b_v(A_v))$. That means that there is no such edge glued to graph $B$ that is incident upon a node that was in $B$ and not in $A$. If one thinks of the reverse direction, we have removed a vertex from $D$ (because according to the “blueprint” $A$ it is not in $C$, but in $B$), but the edge must have come from $C$ (because it is not in $B$). Therefore, there must be an edge without a vertex in $C$, which edge is said to be “dangling”.

• Identification condition: There is no $x, y \in B_v \cup B_E$, such that $x \neq y$, $g(c) = g(y)$ and $y \notin b(A_v \cup A_E)$. It means that two different $A$ (“blueprint” node) should not be mapped to the same node in $B$ object graph if it is deleted.

After creating the categorical background, the graph rewriting methods are outlined, focusing on the algebraic approaches that utilize the formalism presented here.

6.2 Graph Transformation

This section is intended to present the background of the most well-known graph transformation methods. A software engineering-oriented introduction to graph transformation can be found in [Blostein et al. 1995] and [Baresi & Heckel 2002]; a deeper analysis of the topic is included in [Rozenberg 1997] [Ehrig et al. 1999a] [Ehrig et al. 1999b]. An annotated bibliography is collected in [Ehrig and Taenzer 1996].

The first graph transformation methods have evolved from the generalization of Chomsky grammars to non-linear structures like graphs. The earliest initiatives were [Pfaltz & Rosenfeld 1969] [Schneider 1970] [Pratt 1971] [Ehrig 1973]. The major directions are currently the double pushout approach [Ehrig 1973] [Ehrig et al. 1999a], the node-label controlled (NLC) [Janssens & Rozenberg 1980] [Rozenberg, Janssens & Rozenberg 1980] [Rozenberg 1997], the monadic second-order (MSO) logic of graphs [Courcelle 1990]. We have already presented the programmed transformation approach taken by PROGRES in Section 3.3.1.

Graph rewriting is a powerful tool for graph transformations with strong mathematical background. Originally, it was developed as the natural generalization of Chomsky grammars, to generate and parse visual languages. Consequently, there are parallel terms between the two fields. Similarly to term rewriting rules, graph transformation steps are referred to as rewriting rules. Moreover, a graph grammar consists of rewriting rules along with a start graph. A graph grammar with rewriting rules that contain only one node or edge in its left-hand side is called a context-free graph grammar. Unfortunately, in case of graphs, context-free grammars are not as expressive as in parsing textual languages. It has been proven ([Rozenberg 1997], Thrm. 2.4.5 Prop. 1.4.8) that they cannot generate UML class, object, or statechart diagrams, since they are languages containing an infinite amount of square grids.

Instead of the graph language approach, we use the mechanism of the individual parsing steps, the rewriting rules, for graph transformation purposes. The graph transformation is defined as a sequence of rewriting rules, where each such rule is a pair of graphs, called the LHS (left hand side) and RHS (right hand side). The rewriting rule operates as follows: the LHS of a rule is matched against the host graph, forming a redex, and the parts of the redex not included in RHS are removed. The resulting graph is called a context graph.
The next step depends on the rewriting approach that is used. There are two fundamentally different types of adding the elements that occur only in the RHS. In case of the \textit{gluing} approach, the output graph is formed by gluing RHS along the common vertices. The \textit{connecting} approach adds edges to the disjoint union (c.f. Section 6.1) of the context graph and the RHS. To specify the placement of these additional edges, embedding rules are accompanied with the rewriting rule. A typical example for the connecting approach is the NLC rules. The basic difference between the two approaches can be captured by dealing with the context elements. The gluing approach needs the context elements to be specified; the connection approach can work with unknown context. The connection approach, however, cannot preserve elements in the LHS; thus, all the required elements need to be regenerated. The main problem with the context element approach is that it cannot specify rules like \textit{containing all outgoing edges of a node}. In Section 3.4.4 VMTS has provided a solution to how these issues can be treated with multiplicities in metamodel-based transformation rules. Model transformation systems like PROGRES also use a combination of the two approaches.

Since multigraphs can be thought of as a sorted two-structure, the techniques using this formalism are called algebraic approaches. Algebraic graph rewriting provides a way to manipulate objects in a graph category, where the objects are labeled directed graphs and the arrows are graph homomorphisms. There are two main branches of algebraic graph rewriting, namely the double pushout (DPO) and the single pushout (SPO) approaches.

### 6.2.1 The DPO Approach

According to our experience, the DPO approach is one of the most user friendly transformation descriptions for model transformation applications. The main reason for this is the enforcement of the identification condition (c.f. Section 6.1), which is found to be according to the intentions of the transformation modeler. The DPO approach is based on categorical constructs, thus Section 6.3 of this thesis uses categorical formalism in order to be able to reuse the results achieved by the DPO approach. Related to the DPO approach, a rather tutorial like description can be found in [Ehrig 1979] [Ehrig 1987] [Ehrig et al. 1991a] [Corradini et al. 1994], and a more complete summary in [Rozenberg 1997] [Ehrig et al. 1999a]. In this subsection, the main results are summarized and connected to the category theory introduction presented in Section 6.1.

First a graph production rule is defined along with the participants of a rewriting step:

\textbf{Definition 6.9} (graph production):

A graph production rule \( p: (L \xleftarrow{l} K \longrightarrow r R) \) is composed of a production name \( p \), and a pair of injective graph morphism: \( l: K \rightarrow L \), \( r: K \rightarrow R \). The graphs \( L \), \( K \), and \( R \) are called the left-hand side, the interface graph\(^1\) and the right hand side graph of \( p \), respectively.

The DPO approach accomplishes the rule firing through two steps: after finding a redex (the part of the host graph matched by the rule), the first step removes the elements (vertices and edges) from the redex which are in the redex, but not in the RHS graph. This modified redex is referred to as an interface graph. Then, as a second

\(^1\) K stands for the German \textit{Klebegraph} ("gluing graph").
step, the elements of the RHS graph not in the interface graph, but in the RHS graph, are glued to the interface graph.

**Definition 6.10 (direct derivation):**
Given a production rule $p : (L \leftarrow K \rightarrow R)$ and a redex $m : L \rightarrow G$. A direct derivation from $G$ to $H$ using $p$ and based on $m$ exists, if and only if the categorical diagram in Figure 6.5 can be constructed, where both squares are required to be pushouts in the category $\text{Graph}$. It is denoted by $G \Rightarrow H$, or $G \Rightarrow H$, and sometimes $G \Rightarrow H$, if the morphisms are explicitly indicated.

The rewriting rule is characterized by a double pushout (Figure 6.5). The application of the rules results in a direct derivation of the host graph. The category theory framework provides a more flexible and more general background, thus the DPO approach can be applied to many graph-like categories. For labeled and directed graphs, the existence of the pushout (which is the condition to fire a rule) can be ensured by forcing the gluing condition. The gluing condition consists of two parts. Firstly, the identification condition, which states that, on deletion, different vertices in the production rule cannot match the same vertex in the host graph. Secondly, the dangling edge condition has to be dealt with as well: if a vertex should be deleted which is connected to an edge that is not inside the redex, the production rule cannot be fired.

Recalling the means of the pushout construction from Section 6.1, the gluing condition is rather self-explanatory. As far as the left pushout square is concerned, $K$ serves as a “blueprint” to produce $G$, gluing $D$ and $L$ together. In fact, the operation is performed in reverse order, but it is worth noting that this is a categorical equation, where only the relationships (morphisms) are essential. Similarly, based on $K$ as a “blueprint”, $D$ and $R$ are glued together to produce the result $H$; here, the order matches the direction of the arrows.

In the algebraic approaches there is a central topic: the parallel and sequential independence. Parallel independence (Figure 6.6) means that two alternative direct derivations do not conflict, whereas two consecutive direct derivations are sequentially independent if they are not causally dependent. Two alternative direct derivations are parallel independent, if their redexes overlap only in such elements (edges or vertices) that are preserved by both derivations. Two consecutive derivations are sequential independent (Figure 6.7) if the second one is not dependent on the elements added by the first one, and the second one does not delete any items accessed by the first one.
Definition 6.11 (Parallel independence):
Let $G \Rightarrow H_1$ and $G \Rightarrow H_2$ be two direct derivations from the same graph $G$. The two direct derivations are parallel independent, if there exist two graph morphisms $L_1 \xrightarrow{k_2} D_2$ and $L_2 \xrightarrow{k_1} D_1$ such that $l_1 \circ k_2 = m_1$, and $l_1 \circ k_1 = m_2$.

Definition 6.12 (Sequential independence):
Let $G \Rightarrow H_1 \Rightarrow H_2$ be a two-step direct derivation. These derivations are sequential independent if and only if there exist two graph morphisms $R_1 \xrightarrow{k_2} D_2$ and $L_2 \xrightarrow{k_1} D_1$ such that $l_2 \circ k_2 = m_1$, and $l_1 \circ k_1 = m_2$.

Formally, parallel independence means that the overlapping part of the redexes must be in $K_1 \cup K_2$ (i.e. must not be removed by either of the rule), while sequential independence is ensured if and only if the overlapping part of $R_i$ and $R_2$ in $H_1$ must be included in $K_1 \cup K_2$ (i.e. must be preserved).

In the DPO approach, the two independences are equivalent, because the equivalence condition is buried in the gluing condition. To take advantage of this property, we
have chosen the DPO approach as a formal background for our applications. The conditions for parallel and sequential independence are referred to as Local Church-Rosser Problem.

**Theorem 6.1 (Local Church Rosser)**

1. Let $G \Rightarrow H_1$ and $G \Rightarrow H_2$ be two parallel independent direct derivations, as in Figure 6.6; let $m_1' = l_1 \cdot k_1$, where arrow $L_2 \xymatrix{\ar[k_1]^{l_1}\to D_1}$ exists by parallel independence. Then morphisms $\langle l_2, m_2' \rangle$ satisfy the gluing condition; thus, production $p_2$ can be applied to match $m_2'$. Moreover, derivation $G \Rightarrow H_1 \Rightarrow X$ is sequential independent.

2. Let $G \Rightarrow H_1 \Rightarrow H_2$ be a sequential independent direct derivation, as in Figure 6.7; let $m_2' = l_1 \cdot k_1$, where arrow $L_2 \xymatrix{\ar[k_1]^{l_1}\to D_1}$ exists by sequential independence. Then morphisms $\langle l_2, m_2' \rangle$ satisfy the gluing condition; thus, production $p_2$ can be applied to match $m_2'$. Moreover, direct derivations $G \Rightarrow H_1$ and $G \Rightarrow Y$ are parallel independent.

Although we have a condition which makes the parallel and sequential independence interchangeable within the scope of the DPO approach, it can be proven that the statement remains valid if there are no intermediate graphs created when executing the rules in a parallel way. This leads us to the notion of the Parallelism Theorem. But, before stating the theorem, a rule execution method with no intermediate graphs is needed. That is achieved by unifying the rules to be executed parallel with the disjoint union of the two graphs. That construction represents a **truly parallel** rule execution.

**Definition 6.13 (parallel productions)**

Given a graph grammar $G$, a parallel production (over $G$) has the form

$$\langle (p_1, \text{in}^1), \ldots, (p_k, \text{in}^k) \rangle : (L \xymatrix{\ar[l]\leftarrow K \ar[r]\to R})$$

(6.2.1)

(Figure 6.8), where $k \geq 0, p_i : (L_i \xymatrix{\ar[ll]\to K_i \ar[r]\to R_i})$ is a production of $G$ for each $i \in \{1, \ldots, k\}$, $L$ is a coproduct object of the graphs in $\langle L_1, \ldots, L_k \rangle$, and similarly $R$ and $K$ are coproduct objects of $\langle R_1, \ldots, R_k \rangle$ and $\langle K_1, \ldots, K_k \rangle$, respectively. Moreover, $l$ and $r$ are uniquely determined by the families of arrows $\{l_i\}_{i \in k}$ and $\{r_i\}_{i \in k}$, respectively. Finally, each $i \in \{1, \ldots, k\}, \text{in}^i$ denotes the triple of injections $\langle \text{in}_l^i : L_i \to L, \text{in}_K^i : K_i \to K, \text{in}_R^i : R_i \to R \rangle$. Note that all the component productions are recorded in the notation of a parallel production.

![Figure 6.8 Illustration of parallel production](image-url)
Now we can present a theorem central to the DPO approach on the condition of the truly parallel rule execution.

**Theorem 6.2 (Parallelism)**

Given (possibly parallel) productions \( p_1 : (L_1 \xrightarrow{l_1} K_1 \xrightarrow{r_1} R_1) \) and \( p_2 : (L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2) \), the following statements are equivalent:

1. There is a parallel direct derivation \( G \xRightarrow{p_1 \cdot p_2, m} X \).
2. There is a sequential independent derivation \( G \xRightarrow{p_1, m_1, p_2, m_2} H \xRightarrow{p_2, m_2} X \).

Although the DPO approach has several additional results (e.g. embedding, model of computation), only the relevant achievements are discussed in this work.

### 6.2.2 Other Results

Another branch of the algebraic graph transformation approaches is the Single Pushout (SPO) approach [Löwe 1993] [Ehrig & Löwe 1993a] [Korff 1994] [Löwe & Dingel 1994] [Korff & Ribeiro 1995] [Rozenberg 1997]. The single pushout approach uses partial graph homomorphisms to form a single pushout as a rule firing condition in the category of \( \text{Graph}^P \), where labeled directed graphs are the objects, and partial homomorphisms are the arrows. The diagram of the direct derivation according to the SPO approach is depicted in Figure 4.23.

![Figure 6.9 The direct derivation in the SPO approach](image)

The steps of the rewriting are different from that of the DPO approach, because the DPO gluing condition is violated. The conflicts are resolved along the following rule: deletion has priority over preservation.

In the DPO and the SPO approaches, the pushout constructions play an “axiomatic” role. The existence of the pushout is a requirement and all the proofs are built assuming the existence of the characteristic pushout construction. Consequently, the results can be applied to any category, which can exhibit the appropriate pushout diagram. These generalizations are called High-Level Replacement Systems (HLR) [Ehrig et al. 1990] [Ehrig et al. 1991b] [Ehrig & Parisi-Presicce 1992] [Ehrig & Löwe 1993b] [Ehrig et al. 1999b]. The generalization has been applied to especially the DPO approach.
6.3 Extending the DPO Approach to Metamodel-Based Rewriting Rules

In this section, we discuss the topological validation with categorical constructs and provide conditions for the parallel execution of the transformation rules. The closely related publications are [Levendovszky et al. 2004c] [Levendovszky et al. 2005d] [Levendovszky et al. 2005e].

As one can conclude from the previous results, the general metamodel nodes does not characterize the model nodes precisely enough to make well-applicable decisions based on the metamodel. The homomorphic instantiation provides enough information to derive implications for the model based on the metamodel. Consequently, we still have to use the notion of the homomorphic metamodel to validate the rules. As it has been shown in Section 5.2, a compatible homomorphic metamodel can be created with a homomorphization algorithm from the original metamodel. Proposition 5.4 helps to identify those metamodel classes, where the homomorphic metamodel is equivalent to the original one.

Recall from Section 6.2 that the DPO approach specifies an “axiomatic” double pushout diagram, which in actuality is, the rule firing condition. Some preconditions of the double pushout are bound to the rule applications; these are the already elaborated identification and dangling edge conditions (Section 6.1). The first one is related to the rule application algorithm; the second one is a question of the rule formulation. The identification condition can be buried in the algorithm of the specific tool, meaning that it is never violated with injective mapping. It is rarely available as a parameter to adjust for the user of the transformation engine. The dangling edge condition can easily be checked with the following algorithm (Figure 6.10).

\textbf{Figure 6.10} An algorithm for checking the dangling edge condition

As it can be seen in Figure 6.10, the algorithm takes a redex, and a node to be deleted. The algorithm must be invoked for each node to be deleted, checking whether the removal of the node would not cause a dangling edge in the context of the rule. The algorithm is used at rewriting time, since the dangling edge depends not only on the rule, but also on the host graph as well. This notion is illustrated in Figure 6.11.

\textbf{Figure 6.11} A host graph dependent dangling edge condition

The rule matches two nodes and their connecting edge and removes the node \(a\), along with the common edge, leaving behind the node \(b\), denoted with \(b'\) in the RHS graph.
If the match (a-2, b-4) is considered, two dangling edges are left over, namely (1,2) and (2,3). If the match (a-4,b-2) has been found, then the dangling edge condition is satisfied. Based on this example, it can be stated that the dangling edge condition can be checked at rewriting time for certain, assuming the general case.

Knowing the metamodels, however, there is one case when the dangling edge condition can be guaranteed.

**Proposition 6.1**

Let \( O \) be the common ancestor of each elements in the host graph metamodel. If the left hand side of the metamodel-based rewriting rule contains the subgraph depicted in Figure 6.12, then the node \( X \) and the related edges can be removed without violating the dangling edge condition.

\[
\begin{array}{c}
O \quad \quad \quad \quad X
\end{array}
\]

*Figure 6.12 A subgraph for Proposition 6.1.*

**Proof:** Since \( O \) is the ancestor of all classes, it matches all the neighbors of \( X \) with no respect to their type. Consequently, \( O \) has been matched to all nodes adjacent to \( X \). Therefore, each edge incident upon \( X \) is inside the rule. If there are no dangling edges in the rule, then the deletion cannot cause dangling edges. Since LHS and RHS are graphs, they cannot contain dangling edges.

**Remarks.** Obviously, if there is no deletion specified in the rewriting rule, it cannot violate the dangling edge condition. Apart from this trivial case, the one presented in Proposition 6.1 is the only construct considering the metamodel-based rewriting rules, where the dangling edge condition can be guaranteed without knowing the redex.

It is easy to show that there is no such a construct in case of the regular graph rewriting rules. If one takes the LHS of the rewriting rule and connects an edge with a node on the end to each node to be removed, then it always violates the dangling edge condition. This can be performed on an arbitrary rewriting rule.

After examining the conditions related to the means of the rule application, the other parts are taken into consideration. There is always an unknown factor: the host graph, which varies according to the given transformation. As in case of the dangling edge condition, the host graph must be available to validate the rule. This kind of validation is referred to as **online validation**, whereas when the host graph is assumed unknown, it is called **offline validation**. As far as the dangling edge condition is concerned, offline techniques cannot be used, since it is influenced by the host graph to a large extent. Now we turn to the issues that can be validated offline. The advantage of the offline validation is that it need not be checked at transformation time, thus the transformation can be made faster by preprocessing steps. Another important use is to validate the rules at design time and give feedback to the user of the transformation tool. This is similar to the compile time errors in case of textual languages: the compiler filters the erroneous constructs under any circumstances.

The concept of metamodel-based transformation systems offers more information for offline analysis than traditional techniques, since the metamodel of the output graph is also available. The following results benefit from this additional information. For the treatment of the metamodel-based rules, we introduce notations for the...
metamodels. With the help of Figure 6.13, the characteristic double pushout is recalled.

![Double Pushout](image)

**Figure 6.13 The double pushout of the direct derivation**

The metamodels of the participant graphs are denoted with the ‘M’ postfix; thus, they are represented as LM, KM, RM, GM and HM, respectively.

If LM is not partially compatible with GM, then no match is possible. In practice, it is usually prevented by the tools, which offer a list, filled with the metamodel elements of the host graph to select the LM elements. Thus, not satisfying the partial compatibility not only makes no sense, but is usually not possible either. Similarly, if RM is not partially compatible with GM, then no valid output is possible: usually this is banned by the tool. These observations are summarized in a definition:

**Definition 6.13 (Proper metamodels)**

If an LM and an RM satisfy the condition of partial compatibility, elaborated in Proposition 5.4, with respect to GM and HM and they do not contain nonzero multiplicity values, they are called **proper metamodels**.

From now on we will consider examination of proper metamodels only. Since we have a homomorphic metamodel we can use the category theory framework with the `typeof` relationship. Recall that the category `Graph` consists of directed labeled graphs as objects and graph homomorphisms as arrows. Then the categorical diagram in Figure 6.14 can be established for an applicable rule.

![Extended DPO](image)

**Figure 6.14 The DPO extended with the metamodel elements**

The model-metamodel (`typeof`) relationships are denoted with the name of the actual metamodel in lower case: `lm`, `km`, `rm`, `gm`, `dm`, and `hm`. The diagram still raises an issue. If the individual squares form a pushout, what relationships exist between the metamodels? As it has been shown in Proposition 5.4, partial compatibility implies topological inclusion. Consequently, GM contains LM and RM contains RM. This is illustrated in Figure 6.15.
Since KM is the common part of LM and RM, moreover, DM is the intersection of GM and HM, inclusions can be defined between them (Figure 6.16).

As opposed to the inner squares of the diagram, these relationships do not have to be performed, but their possibility must exist. In a sense they are similar to the K interface graph: it is usually not created, but it is always possible to derive it from L and R.

The category \textbf{Set} is a category with sets as objects and functions as morphisms. Graphs can often be treated as two sets and several operations can be performed componentwise. The fact that the diagram made of inclusions in the category \textbf{Set} is a pushout [Barr & Wells 1999] raises the question whether the outer double square is a pushout. In the category \textbf{Set}, this diagram is called Doolittle diagram (Figure 6.17).

If the inner pushout is considered, it can be seen that K is really the intersection of L and D, while G is the union of L and D. Symmetrically, K is the intersection of R and D, whereas H is the union of R and D (if the common nodes and edges are remembered). If the outer square is thought, GM can be computed as the union of LM
and DM, KM is the intersection of DM and LM; KM is the intersection of RM and
DM, and HM is the union of RM and DM. It can be clearly seen that this is true only,
when the following conditions are satisfied for DM (again if the common elements are
traced):

\[
V_{DM} \cap V_{LM} = V_{KM} \\
V_{GM} = V_{DM} \cup V_{LM} \\
V_{DM} \cap V_{RM} = V_{KM} \\
V_{HM} = V_{DM} \cup V_{RM} \\
E_{DM} \cap E_{LM} = E_{KM} \\
E_{GM} = E_{DM} \cup E_{LM} \\
E_{DM} \cap E_{RM} = E_{KM} \\
E_{HM} = E_{DM} \cup E_{RM}
\]

(6.3.1) (6.3.2) (6.3.3) (6.3.4) (6.3.5) (6.3.6) (6.3.7) (6.3.8)

Since the pushout in Graph can be computed componentwise (Section 6.1), the steps
outlined above can be considered as an intuitive proof.

Therefore, the previous line of thought can be summarized as follows.

- As it is well-known from the DPO approach, the inner double square forms
  a double pushout. For each participants of this double pushout, a metamodel
  is provided via a TPM homomorphic mapping.
- Both the meta LDGs and the instance LDGs are objects in Graph.
- Partial compatibilities (lkm, lg, gdm, dhm, rhm, krm), inclusions (l, r, l*, r*,
  m, d, m*) and instantiations (lm, km, rm, gm, dm, hm) are arrows.
- It is suspected that the outer double square is a double pushout.

However, a guarantee is needed:

- The LM and RM must be proper metamodels.
- The conditions (6.3.1)-(6.3.8) must be satisfied.

It is worth noting that if a rewriting rule can be completed, LM and RM are
necessarily proper metamodels; otherwise, they could not have common instances
with GM and HM, respectively.

**Proposition 6.2**

If the morphisms are inclusions in the outer double square (Figure 5.22) and the inner
square is a double pushout, the outer double square (metamodel square) is a double
pushout if the following conditions hold:

\[
V_{DM} = V_{GM} \setminus (V_{LM} \setminus V_{RM}) \\
E_{DM} = E_{GM} \setminus (E_{LM} \setminus E_{RM}) \\
V_{DM} = V_{HM} \setminus (V_{RM} \setminus V_{LM}) \\
E_{DM} = E_{HM} \setminus (E_{RM} \setminus E_{LM})
\]

(6.3.9) (6.3.10) (6.3.11) (6.3.12)

**Proof:** Appendix A.

Based on Proposition 5.4, it can be seen that if the match exists, there is a partial
compatibility between the corresponding metamodels. The properness conditions (c.f.
Definition 6.13) are assumed to be fulfilled for semantic reasons, so there are
inclusion morphisms between the metamodels as it is shown in Figure 5.21. Hence,
the outer double square is a pushout.

It is important to note that the other direction is not always true. If the outer double
square is a double pushout, it is not true that the inner double pushout always exists.
For instance, if LM and RM are proper metamodels, still it is possible that there is no match for a given host graph. At the same time it is possible that this rule can be executed successfully on another host graph. In the first case the inner pushouts could not be created because of the host graph, but the outer pushouts could be composed. However, if the outer pushout cannot be composed, then it is certain that no match can be found, because of the lack of partial compatibility.

Having proven that the outer double square is a pushout, a natural question arises as to whether it is unique up to graph isomorphism. As it was presented in Section 6.2, this can be reduced to the unique existence of the pushout complement object, since the pushout object is always unique, since the pushout is a universal categorical construction. In the category \textbf{Graph} the pushout complement is also unique, provided that the identification and the dangling edge conditions are satisfied. Since the pushout complement \( D \) is shared by both pushout squares, the uniqueness of the pushout complement means that given a host graph and an LHS as well as the redex, the deletion of the unnecessary part and gluing the additional part can be completed in only one way up to isomorphism.

There is another important class of transformations, when the GM and HM metamodels are identical, and the transformation steps do not violate this metamodel. The following proposition gives the mathematical model for this case.

**Proposition 6.3**
If the \( L, K, R, G, D, H \) double square (Figure 6.17) forms a double pushout, and the direct derivation does not violate the metamodel GM, then the following diagrams are pushouts (Figure 6.18).

\[
\begin{align*}
L \leftarrow \downarrow_{kr} & \rightarrow L \\
\downarrow_{k} & \downarrow_{r} \\
GM \leftarrow \downarrow_{dm} & \rightarrow GM \\
\downarrow_{d} & \downarrow_{m}
\end{align*}
\]

\[
\begin{align*}
RM \leftarrow \downarrow_{krm} & \rightarrow RM \\
\downarrow_{k} & \downarrow_{rm} \\
GM \leftarrow \downarrow_{dhr} & \rightarrow GM \\
\downarrow_{d} & \downarrow_{hr}
\end{align*}
\]

**Proof:** Appendix A.

Conforming to the DPO approach results in the “free” propositions given below:

**Proposition 6.4**
If two LM metamodels do not contain elements in common, then the order of the rules are invariant and can be executed in parallel.

**Proof:** Direct consequence of the DPO Parallelism theorem (Section 6.2) applied to the inner double pushout in Figure 6.14.

Proposition 6.4 is especially useful if the transformation environment is expected to offer parallel execution of the rules automatically without collisions and side-effects. Propositions 6.2 and 6.3 facilitate to validate the rewriting rule on the metamodel level without knowing what the actual redex is.
Proposition 6.5
(i) If two rules given by metamodels can be applied sequential independently, the applications can be applied parallel independently as well. (ii) If two rules given by metamodels are parallel independent, the applications are sequential independent as well.

Proof: As can be distilled from the parallelism theorem (Section 6.2) the conditions which make parallel independence and sequential independence interchangeable are buried into the gluing condition (Section 6.2) of the DPO approach. Hence if the same graph always belongs to the metagraph on the instance level, when taking into consideration the sequential and the parallel rule applications, then the proposition follows from the DPO parallelism theorem. That is exactly, however, what the proposition states, since it is about the same application.

6.4 Chapter Summary
Another formal background has been proposed for checking the topological validity of model transformation rules and for providing conditions for parallel execution. The applied mathematical formalism is category theory and the double pushout approach, as it was successfully used in the field of graph rewriting. Since it is required by the category theory, the homomorphic `typeof` mapping is used. A proposition condition for parallel execution has been also proven for the metamodel-based rewriting rules. The main drawback of the approach is that it cannot be applied to those transformation rules which uses completely separate metamodel and model as the result of the rewriting process. Many practical applications do not involve such conditions, thus the results can be used in these transformation steps (for instance the case study in Section 8.2.1).

Future work includes an extension to the SPO approach and performing constraint transformations for homomorphized metamodels.

If novelty is regarded, this is the only solution known to us which combines metamodels and the DPO approach or category theory.
7. Evaluation

In this chapter, the original results are evaluated briefly, based on their applicability and novelty.

Chapter 4 contains novel results that lead to an algorithmic basis for executing metamodel-based rewriting rules. It facilitates the creation of a transformation system, which is really close to UML.

- Propositions have been proven to calculate the number of objects in a valid instantiation of a UML class diagram, as a function of the association multiplicities.
- Based on these propositions, an elimination algorithm has been developed along with a problem-oriented representation. This algorithm helps to reduce the search space of the pattern matching algorithm.
- Heuristics and metamodel-conforming modifications have been performed on the VF2 algorithm to make it more closely fit the requirements of the models extended with metamodel.
- Finally, the refined algorithm has been extended with multiplicity support.

Chapter 5 analyzed the relationship between the model and its metamodel. The goal of the investigation is twofold: first the typeof mapping is examined, then the instantiation. The connection between the metamodel and the model is not close enough, thus a homomorphic metamodel is suggested, on which rule validation can be performed.

- Propositions have proven that the only special metamodels exhibit a homomorphic typeof relationship.
- An algorithm has been suggested to turn a general metamodel to another one that contains homomorphic typeof mappings only. It has been proven that this algorithm generates an equivalent metamodel to the original one, considering the instantiation properties.
- The definitions of compatibility and partial compatibility have been introduced and topological relationships have been shown between compatible and partial compatible metamodels. This facilitates topological validation of the rewriting rules against the metamodels.
- It has been proven that, in general, no homomorphic mapping can be created along the instantiation relationship between the instance graph and the metamodel graph. An algorithm performing the decomposition is provided along with its complexity, which is a polynomial considering the number of edges in the instance graph.

Chapter 6 discussed novel propositions on metamodel-level categorical constructs. The parallelism theorem could be generalized to metamodel level. The propositions in Section 6.3 generalize the achievements of the DPO approach.

- A way has been shown to overcome the restrictions caused by the dangling edge condition of the DPO approach.
- The DPO approach has been generalized to transformation steps, sharing the same metamodels both for the input and output graphs, as well as for the intermediate gluing steps.
- The DPO approach has been generalized to metamodel-level along with the parallelism theorem. Thus, the serial and parallel independence can be exchanged on the metamodel level as well.
8. Application of the Results

In the previous chapters a theoretical background has been developed with the related propositions to underpin a general transformation system. The propositions help to perform validation on the specified rules, they filter out some typical errors and implement such a system providing the algorithmic background.

Books for practitioners (e.g. [OMG MDA] [Mellor & Balcer 2002] [Mellor et al. 2004] [OMG MDA Guide] [Frankel 2003] [Kleppe et al. 2003]) prove that the industry takes interest in MDA concepts. Therefore theoretical results promoting MDA inherently have practical relevance. These achievements, however, together with MDA do not substitute the human creativity and the experienced engineer’s contribution to each individual transformation, directly or incorporated in a generator.

In Chapter 3 the VMTS system was discussed, which is an application of the theoretical results in itself. Since VMTS is also a motivation, in this thesis it was used to introduce the more abstract theoretical approaches from a practical angle. In this chapter some problems are raised and solved illustrating the possible uses of the described transformation system. With the help of these case studies we want to illustrate the following experience:

- Metamodel-based Visual Model Processors are an easy-to-use means to design model transformation and implement MDA model compilers.
- The metamodel-based, uniform treatment of model transformation and designing visual languages are fruitful; the familiarity with UML is enough to use this unified approach.

In Section 8.1 the MDA-related applications are presented, whereas in Section 8.2 the applications target the field of Generative Programming. Due to the generic and flexible nature of VMTS, several problems are solved within this tool, including user interface code generation for Symbian-based mobile devices [Lengyel et al. 2005].

8.1 Model-Driven Architecture

The concepts of MDA were discussed in Section 2.2.3 and 2.3.3 as a main motivation for this work, and an important direction of the software development nowadays. In this section two MDA applications are illustrated, namely, the UML statechart and class diagram compilers.

Considering code generation approaches, there are two main methods to tackle the problem: (i) traversing the model graph, program code is generated for each element, or (ii) provided a graph representation for the code to be generated, graph transformation methods are applied to the statechart graph being transformed into code representation. VMTS offers the possibility for both approaches.

If the traversing method is taken, VMTS facilitates the preparation of Traversing Model Processors. In this case no visual representation is associated with the code, hence it is sometimes quite hard to understand and maintain. The units used in the traversing are nodes and edges; if one wants to group together the semantically attached model parts, a container needs to be created. The real intentions can only be retrieved from the comments or by viewing many lines of program code. This method, however, has distinctly advantageous performance characteristics if the code is written well.

Regarding graph rewriting-based transformation methods, on the other hand, they provide a compensation for their performance penalty. Namely, one can express grouped model elements in a natural way, specifying the left hand side and the right
hand side of the rewriting rule. Thus the conceptually linked model parts are connected together in the transformation description as well. The rule representation offers a visual, diagrammatic perception of the transformation. That is why the model processors built on this approach are referred to as Visual Model Processors in VMTS.

8.1.1 Code Generation for Statecharts with VMTS

In this section, it is shown how to create a Visual Model Processor for VMTS which generates C++ code from statechart models. Because statecharts are widely used in embedded systems, we have decided to choose the Quantum Framework (QF) [Samek 2002], which supports several platforms and operating systems, including those used in embedded and real-time computing. Other approaches also exist, for example [Pintér & Majzik 2005], mainly for formal verifications. The statecharts assumed as an input do not contain AND states. In Figure 8.1 the outline of the transformation process is illustrated.

![Figure 8.1 The overview of the transformation process](image)

The transformation engine requires the metamodel of the statechart diagram, the statechart model from which the code is generated, and the metamodel of the program code representation. The output is the graph representation of the program code, which easily lends itself to a few lines of postprocessing code. Having presented the process as a whole, the individual steps, input and output models are delineated.

8.1.1.1 Representing Statecharts in VMTS

The statechart metamodel is depicted in Figure 8.2. The metamodel specifies an abstract StateElement, which is the base for all elements that can be connected by a directed transition (SynchronizationBar, State). A directed transition is represented by the Transition node. An instance of a Transition node can be connected to two StateElements as the source and target, respectively. A special State element, the OrState, can contain zero or more StateElements. Moreover, there are two special states, namely, the StartState and the EndState.
As an example a simple statechart is depicted in Figure 8.3, which has been taken from [OMG UML 1.5]. The state machine describes the operation of a telephone. Since it includes timed transitions, describing almost every type of event, it is ideal to test the capabilities of the model compiler. The generated code and more details can be downloaded from [VMTS].

In VMTS, attributes are stored in graph labels: the labels are XML documents, and the attributes are represented in an XMI-like format. Figure 6.4 depicts an example for attribute representation, instantiation, and presentation. On the metalevel the properties of the State are described in XML. The following attributes are specified: InstanceName (the name after the instantiation), a string Name property, History, and InternalTransaction. Each property has a multiplicity value, which determines how many attributes can be specified. Furthermore, a description is provided as a help to display the property on the user interface. Figure 8.4 shows not only the XML representation, but also the VMTS GUI offered for the designer.
APPLICATION OF THE RESULTS

The instantiation is accomplished by an XSLT script, which creates an XSD document from the metalevel property XML. The instance level attribute XML must conform to this XSD, otherwise an error message is signaled. The UI displays the members of the grid based on the XSD document, and filling the property XML; it allows only valid attribute specifications, which do not violate conformance to the XSD.

8.1.1.2 Graph Representation for Program Code

In order to apply graph transformation techniques for code generation, the program code itself has to constitute a graph. Fortunately, representing the program code as a graph is quite a natural concept. Compilers use Abstract Syntax Trees (AST) to represent the parsed code. After performing the optimization steps the output code is generated by traversing the tree structure. This concept can also be realized in a higher level: when one wants to create program code for a high level programming language (especially C++, Java or C#), it is possible to offer an appropriate tree structure as the graph representation. Moreover, since there are many similarities in modern object oriented languages, a uniform representation can be created, from which it is possible to generate program code in several languages. The minor differences are handled with code snippets, which are placed directly into the generated code without being processed. A typical example of this construct is the Java Document Model (JDM) and the Abstract Syntax Tree (AST) of the Eclipse JDT Core [Budinsky et al.]. The root of the hierarchy is the CompilationUnit. The CompilationUnit may contain PackageDeclaration nodes. A further child is the TypeDeclaration, which corresponds to the class (or interface) declaration. A class may also incorporate methods (MethodDeclaration), fields (FieldDeclaration) and other programming language constructs. The JDT Core provides many features for debugging and modifying the existing code, but supports Java only.
As far as the multiple language support is concerned the CodeDOM [Dollard 2004] namespaces offer many features, since they are part of the Microsoft .NET Framework, which is inherently a multiple language environment. It is worth noting that these libraries traverse the constructed tree, which means that they use the concepts of traversing model processors. Having chosen C++ as the target language, we have decided to use CodeDOM in our implementation. Since VMTS is a metamodeling environment, a metamodel has been established for the required CodeDOM objects (Figure 8.5).

The CodeTypeDeclaration makes possible the creation of classes and other types (e.g., interfaces, enums). It serves as the basis for the created C++ classes in our case study. With the help of the CodeMemberMethod class, C++ member functions are created and assigned to their classes via the aggregation hierarchy. A member function can have several CodeParameterDeclarationExpressions, which represent a parameter declaration for a method. The general code parts are stored in objects of type CodeSnippetStatement. With the improvement of CodeDOM, the pieces of program code currently assigned to CodeSnippetStatement objects are taken over by objects of other, more specific types. The return values of the functions are supported by the class CodeMethodReturnStatement.

The CodeDOM namespaces contain several other classes; for the case study these provide the necessary code representation background. From the CodeDOM model (which is the result of the transformation) VMTS generates the C# code, which creates the CodeDOM tree from the classes defined in the .NET framework. These are mainly basic tool integration steps that could be eliminated in principle, for instance, using a proprietary framework for the code representation.

8.1.1.3 Transformation Rules

Recall from Section 3.4.4 that firing a graph rewriting production rule involves three main steps: (i) Finding an occurrence of the LHS in the host graph. This means a subgraph of the host graph that is isomorphic to LHS. This subgraph is called match. (ii) Removing the nodes and edges, from the redex graph, which are in the LHS but not in RHS. (iii) Glue the nodes and edges to the redex which are in RHS, but not in LHS. Hence, the host graph is the input model of a transformation step, and the LHS-RHS pairs consist of input model elements. VMTS defines LHS and RHS with the
help of the UML class diagram syntax. Figure 8.6 illustrates the transformation rule for the code generation. According to the rule, a state is selected (lhsState1) together with all its outgoing transitions (lhsSourceTransition, lhsSourceTransition via the object of type lhsTransition) and the targets of the transitions (lhsState2). Then, code is generated for lhsState1, regarding the conditions of transitions and the targets.

As far as the RHS is concerned, for an lhsState1 CodeTypeDeclaration, full-fledged CodeMemberMethod and CodeMemberField objects can be created. Although the metamodel-based rewriting rules solve mostly the topological issues, a question arises: if the RHS elements contain unspecified multiplicity like star, how can the output model be created? More specifically, how many CodeMemberMethod objects should be generated? The answer lies in the attribute transformation process, showing that these transformation aspects are related and dependent on each other. Having found a valid match, the attributes of the matched objects are processed. Currently this step is performed by XSLT scripts. They are assigned to causality relationships pointing from LHS elements to RHS elements. Causality relationships capture the dependencies between the two sides of the rule. These relationships can be one of the following types: (i) creation, (ii) modification, (iii) deletion. Creation results in creating one or more elements of the type specified by the RHS element. The number of the resulting attribute sets defines the number of output graph elements to be created. The second type of the causality relationships describes modifications of existing elements; these are mainly attribute transformation operations. The deletion causality denotes the deletion of the selected LHS node. The VMTS user interface for causalities is also drawn in Figure 8.6. An example for an attribute transformation script can be found in Figure 8.7.
Since the states in the Quantum Framework are implemented as C++ member functions, a new `CodeMemberMethod` needs to be created and filled with the appropriate attributes, one for each state. The result of the transformation for our simple case study can be best viewed in the VMTS Tree View panel (Figure 8.8.)

The containment hierarchy consists of a `QPhone` class definition along with a constructor needed by QF, and states as member functions. The generated code can be found in [VMTS].

### 8.1.2 Code Generation for UML Class Diagram

Our case study is a set of rules which create source code from a UML class diagram. Please recall, from Figure 6.1, that considering the metamodel-based model transformation we need an input and an output metamodel. The input metamodel is the metamodel of the UML class diagram; the output metamodel is the CodeDOM metamodel as it is in the previous case study presented in Section 8.1.1.

Again, the `CodeCompileUnit` is a compilable source of the code graph; it can contain namespaces which are represented by `CodeNamespaces`. `CodeNamespaceImport` allows loading types. `CodeUnits` is an abstract class which can be either a `CodeMemberField` (a class field declaration) or a `CodeMemberMethod` (a class method declaration) or a `CodeTypeDeclaration` (a declaration for class, struct or enumeration). The code member method can contain a `CodeMethodReturnType` (return statement), several `CodeSnippetStatements` (inserts a piece of direct code), and an arbitrary number of `CodeParameterDeclarationExpressions` (the parameters of the function). The attributes for the elements are rather straightforward.
The transformation steps are as follows. Firstly, the UML association relationships must be processed. As usual, they become a reference (or pointer) to another type. This is a preprocessing step, and is executed on the UML class diagram: the associations are mapped to the newly created attributes, the type of these can be either arrays (more than one multiplicity) or simply references (one, zero or one multiplicities). Each association is removed after the first step.

However, this is not the case in our case study: assume that the new attributes violate the metamodel of the class diagram, because only the built-in types can be used to define an attribute. With the flexibility of XML, this issue can be easily solved: the XML schema must be extended with a TemporaryAttribute section; thus, new attributes can be created temporarily and the XSL transformation will be aware of these attributes and process them in the appropriate way. Although this is a flexible way to create intermediate models, these models violate both the input and the output metamodels; thus, they can easily be a source of serious errors, because the rule is not “protected” by the metamodel. To get round this, OCL constraints should be allowed on these attributes.

Figure 8.9 Transformation Rule: Step 2.

Returning to the case study, the second step, which is executed on each preprocessed UML class, is depicted in Figure 8.9. This step accomplishes the conversion step of the transformation: it takes an UML class, and then creates an object of the CodeTypeDeclaration type for the class, along with the objects of four other types (CodeMemberField, CodeMember-Method, CodeParameterDeclarationExpression, and CodeSnippetStatement).

Again, the topology is not really meaningful, because important details are hidden in the attributes. Several multiplicity constraints are undefined (“*”); for instance, the number of CodeMemberMethod depends on the attribute of the given class matched by MetaClass. Thus, these numbers must be provided by the attribute transformation to accomplish the topological rewriting.
APPLICATION OF THE RESULTS

8.2 Generative Programming

Generative programming [Czarnecki & Eisenecker 2000] [Czarnecki et al. 2002] is one of the most pervasive techniques to design and create software systems that are reusable within a specific domain. Reusability is achieved by modeling not only a concrete standalone system (e.g. Rational Unified Process [Kruchten 2003] [Kroll & Kruchten 2003]), but encompassing the whole domain of a field (e.g. a software for cellular telephone family). This domain is modeled by feature models, which have suitable constructs for expressing optional, mandatory, and exclusive properties of certain parts. Feature models also provide notations for constructing a specific system out of all the possible ones. Hence, feature models serve as a guide to compose a concrete required software system [Angyal et al. 2004] [Bisztray et al. 2005].

Unfortunately, the Unified Modeling Language does not directly support the feature modeling notation set. To solve this problem Czarnecki [Czarnecki & Eisenecker 2000] used a generic environment to draw feature models. The Generic
Modeling Environment (Section 3.1) is able to offer support for feature models. GME, however, does not provide visualization opportunities required by feature models such as filled and unfilled angles, and circles at line ends; and there is also no direct model transformation support.

### 8.2.1 Normalizing feature models

To illustrate the novel techniques, a case study is provided in the field of generative programming. Generative programming and domain modeling, together, are one of the most pervasive software technologies. It is applied in fields where the target domain of the software development can be treated, conceived, and captured as one model set with optional, exclusive and inclusive constructs. In the automotive industry, this technology has successfully been used for many years, achieving a wide range of reusability. The first step of capturing the whole domain is the method called feature modeling. Instead of addressing only that part of the domain which is directly relevant to the application to be developed, we model the whole domain for a forthcoming set of applications which are generated on demand. This model contains all the possible results of the engineering process, thus, including the reusability design of the application set. The next steps are to create the generators and the software components for them. The software creation with generators is a highly flexible and safe method, although the effort to create them is higher. Feature modeling must precede the design of the generators, because one must see if the current domain is appropriate for this paradigm. Moreover, feature modeling is used to drive the generator development and to maintain the possible configuration information which serves as a basis for the final composition of the software.

Our case study is a part of this feature-model driven scenario, namely, to transform feature models in a canonical and application specific form, such that it can be fed to the generator directly. Here, we want to illustrate how VMTS works together as a storage system and a transformation engine, thus, the remaining part of the process is beyond the scope of this paper, but we point out that it can be done, for instance, by aspect-oriented constructs, to provide the flexibility to compose a part of the system optionally. There are two preprocessing steps executed by the AGSI transformation engine.

- Normalize the feature diagrams
- For a specific configuration a new feature diagram is constructed with only mandatory features, which is the actual configuration of a system (this can be regarded as the instantiation of the feature model). This diagram will be fed into the generator.

The transformation elaborated here is the first operation. For the in-depth explanation of the normalization process, please refer to [Czarnecki & Eisenecker 2000].

![Figure 8.11 Feature diagram meta-model](image-url)
The metamodel of the feature diagram is shown in Figure 8.11. This metamodel enforces the following properties.

- A node can be a root node (concept) or a child.
- A node can have either XOR (alternative) or OR connector (denoted by arcs).
- A child can be connected to a node of any type (Root or Child), either via these connectors or directly.

Figure 8.12 reveals the first step of normalization: an optional alternative feature can be normalized into a diagram with optional features. This rewriting rule does not change the topology, but updates the feature attributes.

Figure 8.13 presents the second normalization step. A feature with at least one optional child or-feature can be normalized into optional child features.
To show the process itself, we present a simplified version of a feature model for a mobile telephone software system. A mobile telephone can have several types of infra port, but only one of them for the same type. There are, however, several optional parts like a camera, thermometer or Bluetooth: these elements are not necessarily included into a specific telephone type. The diagram appears in Figure 8.14.

![Feature Diagram for Mobile Phone Software](image)

**Figure 8.14 A feature diagram for mobile phone software**

After the normalization process there are only optional features, the alternative feature still holds and the OR relationship is dropped. The result of the transformation is easy-to-follow, as it is shown in Figure 8.15.

![Normalization Process Result](image)

**Figure 8.15 The result of the normalization process**

Another important issue to discuss is sequencing. In the example the rules are executed sequentially, as can be observed in Figure 8.16.
According to the stereotyped activity diagram, Step1 is executed exhaustively (while there is a redex for the LHS of rule in Figure 6.12); then, Step2 (c.f. Figure 8.13 for the corresponding rewriting rule) also runs in an exhaustive manner.

Feature modeling is the part of the VMTS Generative Toolkit, which has been applied in reengineering fixed-route transportation systems [Benedek 2003], and in the automotive industry (within a mentoring cooperation with Robert Bosch GmbH).
Chapter Summary

In this chapter three case studies have been elaborated, two from the field of Model-Driven Architecture and one from the field of Generative Programming. All case studies have illustrated that the metamodel-based rewriting rules are applicable and they can provide simple solutions to quite complex problems. It can also be seen that familiarity with UML is enough to capture the meaning of the transformation rules immediately, which is not easy in case of program code. From these experiences, it can be concluded that the metamodel-based model transformation works well in practice. Since the theoretical results in this thesis have allowed the metamodel-based rules to be executed, this also proves the practical applicability of the results.

The transformation approach implemented in VMTS has been successfully applied to other models of real-world applications. The current research direction is the Graphical User Interface (GUI) modeling with statecharts. Complementary to the well-known resource editors, the recurring dynamic behavior is captured by statecharts, and support for Symbian and .NET platforms are being implemented in VMTS. For the simpler parts both TMP and VMP are being prepared to benefit from the comparison case studies given by industrial applications.
9. Conclusions

9.1 Summary
The results discussed in this work are summarized with four theses. I have proven these results with engineering and mathematical methods and have illustrated their practical relevance in engineering applications. Furthermore, I have published these results in several scientific fora.

Thesis I
Via the Visual Modeling and Transformation System I have shown that
- The class diagram, use case diagram, object diagram, and statechart diagram of the UML 2.0 standard can be realized with n-layer metamodeling techniques with only one instantiation relationship, namely, the one between MOF M0-M1 layers.
- Feature models and resource models can also be realized by these techniques.
- The UML class diagram can be used as a language for the LHS and the RHS in the transformation rules: this construct can be applied to engineering applications.
- Unifying the metamodeling and the graph transformation techniques creates a useful combination in model transformation systems.

Thesis II
I have given a pattern matching algorithm for the left-hand-side of the metamodel-based rules, along with an algorithm that provides conditions to split the search space of the pattern matching algorithm.
- The class diagram depicted in Figure 4.8 can be instantiated by $na'$ objects of type A and $nb'$ objects of type B, where $n$ is an arbitrary positive integer and

$$a' = \frac{a}{\gcd(a,b)} \quad (9.1.1)$$
$$b' = \frac{b}{\gcd(a,b)} \quad (9.1.2)$$

where $\gcd$ denotes the greatest common divisor.
- If no multiple edges are allowed for associations in the object diagram, the class diagram in Figure 4.8 can be instantiated by $na$ objects of type A and $nb$ objects of type B, where $n$ is an arbitrary positive integer.
- The worst case number of objects that must be examined in order to decide the valid instantiation is in general

$$\#A + \#B \quad (9.1.3)$$

and the greatest value of the corresponding number of objects:

$$n = \min \left[ \frac{\#A}{a}, \frac{\#B}{b} \right] \quad (9.1.4)$$

In Formulae (9.1.3) and (9.1.4) $\#A$ denotes the number of A objects, and $\#B$ denotes the number of B type objects.
- In the IMM algorithm, the matrix resulted through an arbitrary elimination step represents an equation set which is equivalent to the multiplicity
equations established according to the instantiation equation. Furthermore there is no solution which can form a valid instantiation, but it is not among the results of the IMM algorithm.

- The worst case of the `compute_candidate_pairs()` procedure (Figure 4.16) is the one provided by the VF2 algorithm (Figure 4.4), assuming connected pattern graph.
- If we prefer to choose the $V^{LHS}$ metaelement having the most instances in the next step of the algorithm, the termination of the algorithm can require more steps than the `MetaVF2` algorithm.

**Thesis III**

I have supplied a method for the topological validation of the metamodel-based rules assuming that the metamodels of the input model and the output model are available.

- Let $Meta$ be an LDG with labels conforming to the UML class diagram. Let another LDG $Instance$ be given with labels conforming to the UML object diagram. Furthermore, we assume that $Instance$ instantiates $Meta$, according to the instantiation rules enforced by UML. If $Meta$ does not contain inheritance relationship (nor abstract classes, which would be semantically meaningless without inheritance), then the TPM between the $Instance$ and $Meta$ graph is a graph homomorphism.

- Let $Meta$ be an LDG with labels conforming to the UML class diagram. Let another LDG $Instance$ be given with labels conforming to the UML object diagram. In addition we assume that $Instance$ instantiates $Meta$ according to the instantiation rules enforced by UML. $Instance$ can be mapped to $Meta$ via graph homomorphism if and only if no $O$ object is attached to link from more than one $L_n$ set, where $A_n$ belongs to the class hierarchy of $C$, which is the type of $O$.

- The homomorphic metamodel and its generator metamodel are equivalent.

- Let $Meta1$ and $Meta2$ be an LDG, with labels conforming to the UML class diagram, both homomorphic metamodels. If $Meta1$ is compatible with $Meta2$, $Meta1$ is a (not always connected) subgraph of $Meta2$ not regarding the multiplicity labels. The following formula is always true:

$$\text{Meta1M} \subseteq \text{Meta2M}$$  \hspace{1cm} (9.1.5)

where $M_{Meta1}$ and $M_{Meta2}$ are the sets of the allowed multiplicity for the $Meta1$ and $Meta2$ side respectively at the same topological position.

In case of partial compatibility, where the zero multiplicity values are not allowed, it is enough to enforce the following conditions for each corresponding multiplicity pairs:

$$\text{SupM}_{Meta1} \leq \text{SupM}_{Meta2}$$  \hspace{1cm} (9.1.6)

$$M_{Meta1} \cap M_{Meta2} \neq \emptyset$$  \hspace{1cm} (9.1.7)

where $\text{Sup}$ is the supremum (least upper bound) of the set which contains the allowed multiplicity values.

- Each $I$ instance graph can be decomposed into $I_1, I_2 \ldots I_K$ non-isomorphic subgraphs such that the instantiation relationship between the homomorphic $M$ metagraph having nonzero multiplicities on the association ends; the $I_i, (k = 1..K)$ graph is a graph homomorphism, where $I$ is an instance of $M$. 


The worst case complexity of the decomposition algorithm is
\[ O(\#E(I)\#E(M)). \] (9.1.8)

**Thesis IV**
I have provided conditions for the parallel rule applications for metamodel-based rules, and for changing their order without altering the result of the transformation.

- Let \( O \) be the common ancestor of each element in the host graph metamodel. If the metamodel-based rewriting rule contains the subgraph depicted in Figure 6.12, then the node \( X \) and the related edges can be removed without violating the dangling edge condition.

- I have supplied a categorical model for the metamodel-based transformation rules. Furthermore, I have shown that if the morphisms are inclusions in the outer double square, and the inner square is a double pushout, the outer double square is a double pushout if the following conditions hold:
  \[ V_{DM} = V_{GM} \setminus (V_{LM} \setminus V_{RM}) \] (9.1.9)
  \[ E_{DM} = E_{GM} \setminus (E_{LM} \setminus E_{RM}) \] (9.1.10)
  \[ V_{DM} = V_{HM} \setminus (V_{RM} \setminus V_{LM}) \] (9.1.11)
  \[ E_{DM} = E_{HM} \setminus (E_{RM} \setminus E_{LM}) \] (9.1.12)
If the \( L, K, R, G, D, H \) double square (Figure 6.16) is a double pushout, and the direct derivation does not violate the metamodel GM, then the diagrams in Figure 6.18 are pushouts.

- If two LHS metamodels do not contain elements in common, then the order of the rules are invariant and can be executed parallel.

- (i) If two rules given by metamodels can be applied sequentially independently, the applications can be applied parallel independently as well. (ii) If two rules given by metamodels are parallel independent, the applications are sequential independent as well.

**9.2 Future Work**
Future work includes several directions: some of them have already been given during the evaluation. This section summarizes the main areas of future research.

- Optimizing the match algorithm with more efficient heuristics, and more compact generators generated for specific models. There are cases when theoretical proofs cannot be proven, because there is no general truth that holds for each model. There are, however, special properties of the models that hold frequently, but not always. These situations require proof by simulation to customize the matching algorithm to software model transformation applications. These measurements are planned to investigate these acceleration facilities. Also, these activities can show the acceleration rate provided by the heuristics in this thesis. Further optimization of the IMM algorithm.

- Constraint-based methods for validating transformation steps. This is one of the most promising directions of future research. The metamodel-based rule specification allows borrowing more concepts from UML, namely, the Object Constraint Language (OCL). Using constraints in OCL, one can ensure that specific model properties can be guaranteed, validated and preserved during the transformation. If these constraints are specified correctly, and the transformation completes successfully, one can be
certain that these properties were handled correctly during the transformation process.

- Eliminating the crosscutting concerns from the constraint assignments: applying Aspect-Oriented methods with weaving algorithms.
- Generalizing the SPO approach to metamodels. The limitations of the DPO approach can be observed in this work. Despite the mentioned drawbacks of the SPO approach, in our opinion, it is worth an experiment, whether there are applications where the SPO approach has more advantageous properties over the DPO approach. Investigating the generalization possibilities in high-level replacement systems.
- Using proposition 6.3 and 6.4 we can simplify the constraint checking algorithm for the rewriting rule.
- Conditions for rule terminations, and the correctness criteria: the transformation guarantees the conformance of the result to the output metamodel, based on the rules only.
- Implementing a cluster-based edition of VMTS, where the parallel rule applications can be applied.
- Developing further MDA model compilers. The main pillars of the automated code generation, the class and state diagrams have been successfully addressed by metamodel-based transformation methods. Other applications are also needed to be supported. Mobile software development with combined statechart and UI modeling has real practical relevance, considering Symbian, Linux [Bányász & Levendovszky 2003] Microsoft, and J2ME mobile platforms.
- Investigating the possible relations to other fields, where the metamodel-based modeling and transformation principles can be applied or vice versa. For instance, graph mining [Iváncsy et al. 2005] can accelerate the pattern matching in VMTS in case of large models. VMTS can serve as a framework for performance analysis tools [Bogárdi-Mészöly et al. 2005]

The results of this thesis are recent results of a relatively new discipline. This means that the field of software model transformation, including these contributions, must go through many adjustments, practical experiences to obtain real practical relevance in the industry. Hopefully, the achievements of this thesis will promote, at least to an infinitesimal extent, this process and advance more efficient automation of the software engineering industry.
A. Detailed Proofs

Proof for Proposition 6.2
As it is known, the category $\text{Graph}$ has all pushouts, which means that for every pair of morphism there is a pushout in the category. In Section 6.2 it has been elaborated that if given two morphism $b: A \to B$, $c: A \to C$, then the pushout $\langle D, g: B \to D, f: C \to D \rangle$ can be constructed as $D = (B_v + C_v) \approx (B_e + C_e) \approx$ i.e. the quotient set of the disjoint union modulo $\approx = \{f(a), g(a) | a \in A\}$ smallest equivalence relation that maps each element to its equivalence class. For $\text{Graph}$, the pushout construction is accomplished componentwise: for the edge set and the vertex set, respectively.

First we create the disjoint union:

$$V_{GM, separated} = V_{LM} + V_{DM} \quad (A.1)$$

Since the relationship between GM and LM is an inclusion (LM is a proper metamodel), GM can be decomposed to LM and another graph ($V_{XM}$):

$$V_{GM} = V_{XM} \cup V_{LM} \quad (A.2)$$

DM is created from LM by removing the parts that are in LM but not in RM:

$$V_{DM} = V_{XM} \cup V_{LM} \setminus (V_{LM} \setminus V_{RM}) \quad (A.3)$$

KM is created similarly from LHS:

$$V_{KM} = V_{LM} \setminus (V_{LM} \setminus V_{RM}) \quad (A.4)$$

From (A.3) and (A.4):

$$V_{DM} = V_{XM} \cup V_{KM} \quad (A.5)$$

Rearranging Eq. (5.3.16):

$$V_{LM} = V_{KM} \cup (V_{LM} \setminus V_{RM}) \quad (A.6)$$

From Equations (A.1), (A.5) and (A.6):

$$V_{GM, separated} = V_{LM} + V_{DM} = (V_{XM} \cup V_{KM}) + (V_{KM} \cup (V_{LM} \setminus V_{RM})) \quad (A.7)$$

The mapping for the equivalence is provided by KM, the other nodes are glued:

$$(V_{DM} + V_{LM})_v = V_{XM} \cup V_{KM} \cup V_{LM} \setminus V_{RM} \quad (A.8)$$

Substituting Eq. (A.6), the pushout object is as follows:

$$(V_{DM} + V_{LM})_v = V_{XM} \cup V_{LM} = V_{GM} \quad (A.9)$$

The same derivation must be done for the edge sets:

$$E_{GM, separated} = E_{LM} + E_{DM} \quad (A.10)$$

$$E_{GM} = E_{XM} \cup E_{LM} \quad (A.11)$$

$$E_{DM} = E_{XM} \cup E_{LM} \setminus (E_{LM} \setminus E_{RM}) \quad (A.12)$$

$$E_{KM} = E_{LM} \setminus (E_{LM} \setminus E_{RM}) \quad (A.13)$$

$$E_{DM} = E_{XM} \cup E_{KM} \quad (A.14)$$

$$E_{LM} = E_{XM} \cup (E_{LM} \setminus E_{RM}) \quad (A.15)$$

$$E_{GM, separated} = E_{LM} + E_{DM} = (E_{XM} \cup E_{KM}) + (E_{KM} \cup (E_{LM} \setminus E_{RM})) \quad (A.16)$$

$$(E_{DM} + E_{LM})_v = E_{XM} \cup E_{KM} \cup E_{LM} \setminus E_{RM} \quad (A.17)$$

$$(E_{DM} + E_{LM})_v = E_{XM} \cup E_{LM} = E_{GM} \quad (A.18)$$

The pushout must also be composed for the gluing direction:
\[ V_{HM,\text{separated}} = V_{RM} + V_{DM} \] (A.19)

\[ V_{HM} = V_{YM} \cup V_{HM} \] (A.20)

\[ V_{DM} = V_{YM} \cup (V_{RM} \setminus (V_{RM} \setminus V_{LM})) \] (A.21)

\[ V_{KM} = V_{RM} \setminus (V_{RM} \setminus V_{LM}) \] (A.22)

\[ V_{DM} = V_{YM} \cup V_{KM} \] (A.23)

\[ V_{RM} = V_{KM} \cup (V_{RM} \setminus V_{LM}) \] (A.24)

\[ V_{HM,\text{separated}} = V_{RM} + V_{DM} = (V_{YM} \cup V_{KM}) \cup (V_{KM} \cup (V_{RM} \setminus V_{LM})) \] (A.25)

\[ (V_{DM} + V_{RM})_s = V_{YM} \cup V_{KM} \cup V_{RM} \setminus V_{LM} \] (A.26)

\[ (V_{DM} + V_{RM})_s = V_{YM} \cup V_{RM} = V_{HM} \] (A.27)

Computing the pushout object of the edge set:

\[ E_{HM,\text{separated}} = E_{RM} + E_{DM} \] (A.28)

\[ E_{HM} = E_{YM} \cup E_{HM} \] (A.29)

\[ E_{DM} = E_{YM} \cup E_{RM} \setminus (E_{RM} \setminus E_{LM}) \] (A.30)

\[ E_{KM} = E_{RM} \setminus (E_{RM} \setminus E_{LM}) \] (A.31)

\[ E_{DM} = E_{YM} \cup E_{KM} \] (A.32)

\[ E_{RM} = E_{KM} \cup (E_{RM} \setminus E_{LM}) \] (A.33)

\[ E_{HM,\text{separated}} = E_{RM} + E_{DM} = (E_{YM} \cup E_{KM}) \cup (E_{KM} \cup (E_{RM} \setminus E_{LM})) \] (A.34)

\[ (E_{DM} + E_{RM})_s = E_{YM} \cup E_{KM} \cup E_{RM} \setminus E_{LM} \] (A.35)

\[ (E_{DM} + E_{RM})_s = E_{YM} \cup E_{RM} = E_{HM} \] (A.36)

**Proof for Proposition 6.3**

\[ V_{GM,\text{separated}} \] can be written as follows:

\[ V_{GM,\text{separated}} = V_{LM} + V_{GM} \] (A.37)

If at least one direct derivation is possible, then the partial compatibility is satisfied on the instance level. It means that GM can be decomposed:

\[ V_{GM} = V_{XM} \cup V_{LM} \] (A.38)

(A.37) and (A.38) then suggests:

\[ V_{GM,\text{separated}} = V_{LM} + (V_{XM} \cup V_{LM}) \] (A.39)

Since LM is the basis of the equivalence relation, we have:

\[ V_{GM,\text{separated}} = V_{XM} \cup V_{LM} = V_{GM} \] (A.40)

We need to prove the proposition for the edge set objects as well:

\[ E_{GM,\text{separated}} = E_{LM} + E_{GM} \] (A.41)

\[ E_{GM} = E_{XM} \cup E_{LM} \] (A.42)

\[ E_{GM,\text{separated}} = E_{LM} + (E_{XM} \cup E_{LM}) \] (A.43)

\[ E_{GM,\text{separated}} = E_{XM} \cup E_{LM} = E_{GM} \] (A.44)

The other pushout can be proven similarly:
\[ V_{HM, \text{separated}} = V_{RM} + V_{HM} \]  \hfill (A.45)

\[ V_{HM} = V_{YM} \cup V_{RM} \]  \hfill (A.46)

\[ V_{HM, \text{separated}} = V_{RM} + (V_{YM} \cup V_{RM}) \]  \hfill (A.47)

\[ V_{HM, \text{separated}} = V_{YM} \cup V_{RM} = V_{HM} \]  \hfill (A.48)

\[ E_{HM, \text{separated}} = E_{RM} + E_{HM} \]  \hfill (A.49)

\[ E_{HM} = E_{YM} \cup E_{RM} \]  \hfill (A.50)

\[ E_{HM, \text{separated}} = E_{RM} + (E_{YM} \cup E_{RM}) \]  \hfill (A.51)

\[ E_{HM, \text{separated}} = E_{YM} \cup E_{RM} = E_{HM} \]  \hfill (A.52)
### B. Comparison Table for Model Transformation Tools

<table>
<thead>
<tr>
<th>Rule specification</th>
<th>VMTS</th>
<th>GReAT</th>
<th>AGG</th>
<th>PROGRES</th>
<th>VIATRA</th>
<th>ATOM3</th>
<th>FUJABA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Metamodel-Based</td>
<td>Proprietary Pattern Language with Type Information</td>
<td>Instance Graph with type information</td>
<td>Proprietary Pattern Language with Type Information</td>
<td>Proprietary Pattern Language with Type Information</td>
<td>Typed DPO rules</td>
<td>Story diagrams</td>
</tr>
<tr>
<td>Model Specification</td>
<td>Metamodel+model</td>
<td>Metamodel+model</td>
<td>Model + schema</td>
<td>Metamodeling with proprietary instantiation (VPM)</td>
<td>Metamodel+ model, n-layer metamodeling with UI specification</td>
<td>Two layer metamodeling with class diagram syntax</td>
<td></td>
</tr>
<tr>
<td>Attribute Specification</td>
<td>XML</td>
<td>UDM (Proprietary)</td>
<td>JAVA</td>
<td>Proprietary</td>
<td>Graph representation, built-in support for basic JAVA Types</td>
<td>Python</td>
<td>Metamodel translated into JAVA</td>
</tr>
<tr>
<td>Constraints in model</td>
<td>Instantiation + OCL</td>
<td>OCL interpreter</td>
<td>Visual constraints with visual pre- and postconditions + JAVA + instantiation</td>
<td>Proprietary integrity constraints</td>
<td>Graph pattern with arbitrary levels of negation</td>
<td>Specified in Python, checked on events. The constraints can be pre- and postconditions to events. Semantic and graphical constraints.</td>
<td>OCL (MOF 2.0 Plugin)</td>
</tr>
<tr>
<td>Constraints in the rule</td>
<td>Instantiation + OCL</td>
<td>OCL</td>
<td>JAVA, NAC</td>
<td>Attribute constraints, cardinality, negative edge</td>
<td>Graph pattern</td>
<td>Generalized NAC, application conditions.</td>
<td>Story diagram, JAVA</td>
</tr>
<tr>
<td>Control flow</td>
<td>Stereotyped activity diagrams</td>
<td>Deterministic, non-deterministic, recursion</td>
<td>Layers (exhaustive or once, loop)</td>
<td>Imperative and declarative, transactions</td>
<td>ASM</td>
<td>Layers with priorities, sequencing by priority. Parallel execution of non-overlapping matches.</td>
<td>Story diagrams</td>
</tr>
<tr>
<td>Attribute transformation</td>
<td>XSL</td>
<td>C-based attribute mapping language</td>
<td>JAVA</td>
<td>Built-in or host programming language (esp. C)</td>
<td>ASM statements (built in support for basic JAVA types)</td>
<td>Python</td>
<td>Story diagrams</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>MetaVF2 with heuristics</td>
<td>Edge-based iteration</td>
<td>CSP, CSP w/o BJ, Simple BT</td>
<td>Plan-based</td>
<td>CSP (interpreter), Model-specific local search (plugin)</td>
<td>Taken from [Dörr 1995]</td>
<td>Local search with BFS search plan (no backtrack)</td>
</tr>
<tr>
<td>Primary application area</td>
<td>Software model transformation</td>
<td>Software model transformation</td>
<td>Type-aware graph transformation</td>
<td>Graph transformation</td>
<td>Formal model analysis</td>
<td>Simulation</td>
<td>UML and JAVA round-trip engineering</td>
</tr>
</tbody>
</table>
Bibliography


[Assman 2000] Assman U: Graph rewrite systems for program optimization, ACM Transactions on Programming Languages and Systems 22(4), 2000, pp. 583-637


[Bányász & Levendovszky 2003]


http://www.cs.york.ac.uk/puml/mmf/mmf.pdf


[FUJABA] FUJABA (From Uml to Java And Back Again) Homepage http://www.cs.upb.de/cs/fujaba/.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lengyel &amp; Levendovszky 2004</td>
<td>Lengyel L, Levendovszky T: Aspektus-orientált programozás, Híradástechnika, Volume LIIX, 2004/10, pp. 8-12</td>
</tr>
<tr>
<td>Lengyel et al. 2004a</td>
<td>Lengyel L, Levendovszky T, Charaf H: Constraint Handling in Feature Models, 5th International Symposium of Hungarian Researchers on Computational Intelligence, Budapest, 2004</td>
</tr>
</tbody>
</table>


[Lengyel et al. 2005i] Lengyel L, Levendovszky T, Charaf H: A Case Study: Supporting Metamodel-Based Graph Transformation with Aspect-Oriented Constraints, 5th Internal Conference of PhD Students, Miskolc, Hungary, 14-20 August 2005


[Levendovszky & Charaf 2005] Levendovszky T, Charaf H: Pattern Matching in Metamodel-Based Model Transformation Systems, Accepted to Periodica Polytechnica Electrical Engineering, 2005


[Mens et al. 2005] Mens T, Gorp PV, Varró D, Karsai G: Applying a Model Transformation Taxonomy to Graph Transformation Technology, GraMoT’05 - International Workshop on Graph and Model Transformation, Tallinn, 2005


[OMG CP] UML Profile for CORBA

[OMG EAI] UML Profile for Enterprise Application Integration (EAI)

[OMG EDOC] UML Profile for Enterprise Distributed Object Computing (EDOC)


[OMG TP] UML Testing Profile http://www.uml.org/


[PROGRES] The PROGRES system can be downloaded from http://mozart.informatik.rwth-aachen.de/research/projects/progres/main.html


[PUML] The Precise UML Group Homepage www.puml.org


[Rosello 2004] Rossello F: Chemical Graphs, Chemical Reaction Graphs, and Chemical Graph Transformation, International Workshop on Graph-Based Tools (GraBaTs), Rome, Italy, 2004


<table>
<thead>
<tr>
<th>Reference</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>[VMTS]</td>
<td><a href="http://avalon.aut.bme.hu/~tihamer/research/vmts/">http://avalon.aut.bme.hu/~tihamer/research/vmts/</a></td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>