TRANSFORMATION-BASED SUPPORT FOR VISUAL LANGUAGES

Ph.D. Thesis

Gergely Mezei

Advisors:
Dr. Hassan Charaf, Ph.D.
Associate Professor

Dr. Tihamér Levendovszky, Ph.D.
Senior Lecturer

Budapest, 2007
# Table of Contents

List of Figures vi

List of Tables viii

Acknowledgments xii

Chapter 1
Introduction 1
1.1 Thesis Contributions ............................................. 2
1.2 Thesis Structure .................................................... 4

Chapter 2
Motivation 6
2.1 Modeling .............................................................. 7
2.1.1 The Unified Modeling Language ................................. 7
2.1.2 Domain-Specific Modeling ....................................... 8
2.1.3 Metamodeling ...................................................... 8
2.1.4 Instantiation ....................................................... 9
2.1.5 Constraints ....................................................... 10
2.2 Model Transformation ............................................... 11
2.3 Model-Driven Engineering .......................................... 12
2.3.1 Model-Driven Architecture .................................... 12
2.3.2 Model Integrated Computing ................................... 13
2.4 Open Issues ......................................................... 13
2.5 Visual Modeling and Transformation System – Overview .......... 14
2.6 Chapter Summary ..................................................... 15

Chapter 3
Theoretical Background 16
3.1 Abstract State Machines ........................................... 16
3.1.1 Introduction to ASM ............................................ 16
3.1.2 Mathematical Definition of ASM .............................. 18
3.2 Category Theory .................................................... 18
3.3 Algebraic Graph Transformation Approaches ..................... 20
3.3.1 The DPO Approach ............................................ 21
3.3.2 Parallelism Theorem ............................................ 21
3.3.3 Concurrency Theorem ........................................... 23
# Table of Contents

3.3.4 Embedding and Amalgamation ........................................ 24  
3.3.5 Termination Criteria ............................................... 25  
3.3.6 Model Transformations in VMTS ................................... 26  
3.3.6.1 Transformation Control Flow ................................ 26  
3.3.6.2 Specialities in Rewriting ..................................... 26  
3.4 Chapter Summary ..................................................... 27  

Chapter 4  
Defining Visual Languages .................................................. 29  
4.1 Backgrounds and Related Work ........................................ 30  
4.1.1 Attributes of Model Elements ..................................... 30  
4.1.1.1 Existing Approaches .......................................... 30  
4.1.1.2 Deep Instantiation ........................................... 31  
4.1.1.3 Representing the Attributes .................................. 31  
4.1.2 Concrete Syntax ................................................... 32  
4.1.2.1 Classifying Concrete Syntax Definition Approaches ....... 32  
4.1.2.2 Existing Tools ............................................... 34  
4.2 N-Layer Attribute Structure .......................................... 36  
4.2.1 Attribute Structure ................................................ 36  
4.2.2 The Simplified N-Layer Attribute Algebra ....................... 36  
4.2.3 Valid and Invalid Models ......................................... 39  
4.2.4 The Transformation ............................................... 40  
4.2.5 Supporting Deep Instantiation .................................... 41  
4.2.5.1 Extending the Attribute Algebra ............................. 41  
4.2.5.2 Extending the Instantiation Transformation ................ 43  
4.2.6 Properties of the Enhanced N-Layer Attribute Algebra ..... 44  
4.3 Defining the Concrete Syntax ......................................... 45  
4.3.1 A Metamodel for Visualization .................................... 46  
4.3.2 A Model Transformation for Processing the Concrete Syntax 47  
4.3.3 Properties of the Transformation ................................ 49  
4.4 Chapter Summary ..................................................... 50  

Chapter 5  
Optimized Metamodel-Based Constraint Validation ...................... 51  
5.1 Backgrounds and Related Work ...................................... 52  
5.1.1 Theoretical Results on OCL Formalization ....................... 52  
5.1.2 Tool Support for OCL ............................................ 53  
5.2 Constraint Validation ............................................... 54  
5.2.1 Extending OCL .................................................... 54  
5.2.1.1 Supporting the N-layer Attribute Algebra .................. 54  
5.2.1.2 Supporting Attributed Relations .............................. 55  
5.2.2 OCLASM: A New Formalism for OCL ............................. 56  
5.3 Constraint Optimization ............................................. 63  
5.3.1 Constraint Relocation ............................................ 63  
5.3.1.1 Restrictions ................................................ 64  
5.3.1.2 Formalizing the Relocation ................................. 71  
5.3.1.3 The Relocation Algorithm ................................... 74
### TABLE OF CONTENTS

5.3.2 Constraint Caching ........................................... 75
5.3.2.1 The Algorithm ........................................ 75
5.3.2.2 Formalizing the Algorithm ............................. 78
5.4 Chapter Summary .............................................. 79

Chapter 6
Truly Parallel Graph Transformation 80
6.1 Backgrounds and Related Work ............................... 80
6.1.1 Acceleration of Graph Rewriting .......................... 81
6.1.2 Model Transformation Approaches ......................... 81
6.2 Parallel Model Transformations ............................... 82
6.2.1 Basic Definitions ........................................ 83
6.2.2 Transformation-Level Parallelism ........................ 84
6.2.2.1 Dependency Between the Rules ......................... 84
6.2.2.2 Handling Possibly Conflicting Rules .................. 86
6.2.3 Rewriting Rule-Level Parallelism ........................ 89
6.2.3.1 Refining the Maximum Amount of Steps ............... 89
6.2.3.2 Parallelization Based on Fix Pivot ................... 90
6.2.3.3 Parallel Matching Based on Pseudo Random Paths .... 91
6.2.4 Composing the Truly Parallel Graph Transformation Approach .... 93
6.2.4.1 Initialization and Matching ............................ 94
6.2.4.2 Finishing the Match, Rewriting ....................... 95
6.2.4.3 Stepping the Transformation ........................... 96
6.2.5 Performance Measurements ............................... 97
6.2.5.1 Performance Factors .................................. 97
6.2.5.2 The Case Study ....................................... 97
6.2.5.3 Test Results .......................................... 98
6.3 Chapter Summary .............................................. 101

Chapter 7
Application of the Results 103
7.1 The Visual Modeling and Transformation System ............ 103
7.1.1 Modeling Framework ...................................... 104
7.1.1.1 VMTS Presentation Framework ......................... 104
7.1.1.2 Visual Appearance Definition ........................ 108
7.1.2 Extending, Optimizing Constraint Handling ............... 110
7.1.2.1 OCL Compiler ....................................... 110
7.1.2.2 Compiler Extensions ................................ 111
7.1.3 Model Transformation ..................................... 112
7.1.3.1 AGSI Compact Framework ........................... 112
7.1.3.2 VMTS Parallel Transformation Engine ............... 113
7.2 Model-Based Development with VMTS ....................... 115
7.2.1 Support for UML Diagrams ............................... 115
7.2.2 Graphical Programming Languages ....................... 116
7.2.3 Network Modeling ........................................ 117
7.2.4 Constraint Optimization ................................ 119
7.2.5 Other Applications ..................................... 121
TABLE OF CONTENTS

7.3 Chapter Summary ................................................................. 121

Chapter 8

Conclusions 123
8.1 Summary ................................................................. 123
8.2 Application of the Theoretical Results .......................... 126
8.3 Future Work ................................................................. 128

Appendix A

Detailed Proofs 130
# List of Figures

2.1 The basic concepts of MIC [Sztipanovits and Karsai, 1997] .......................... 13  
3.1 A diagram representation of pushout .................................................. 19  
3.2 Production rule in the DPO approach ................................................. 21  
3.3 Parallel independence ........................................................................... 22  
3.4 Sequential independence ....................................................................... 22  
3.5 E-dependency production ....................................................................... 23  
3.6 E-related transformation ......................................................................... 24  
3.7 Concurrency Theorem ............................................................................ 24  
3.8 Example control flow using VCFL ......................................................... 26  
4.1 The modeling layers in MOF ................................................................... 30  
4.2 Metamodel used in Function modeling .................................................. 33  
4.3 Concrete Syntax Definition - Overview ................................................. 45  
4.4 VMTS Presentation DSL - Metamodel .................................................. 46  
4.5 Model Transformation in VMTS - Overview .......................................... 48  
4.6 VMTS Presentation DSL - Control Flow ............................................. 48  
4.7 The rule MatchAppearances ................................................................... 49  
5.1 The steps and the whole RelocationPath ............................................... 64  
5.2 Example metamodel and model .............................................................. 65  
5.3 ExactlyOne multiplicity on both sides - metamodel and model ............... 66  
5.4 Zero multiplicity - metamodel and model .............................................. 67  
5.5 MoreThanOne → ExactlyOne multiplicity - metamodel and model ......... 68  
5.6 ExactlyOne → OneOrMore multiplicity - metamodel and model ............ 70  
6.1 Blocks and metaconflicts ....................................................................... 88  
6.2 The hierarchy of the parallel model transformation approach .................. 94  
6.3 The sequence diagram of the communication ......................................... 95  
6.4 The sequence diagram of the communication (2) ................................... 96  
6.5 Case study - example model and malicious configuration ....................... 98  
7.1 The structure of VMTS .......................................................................... 104  
7.2 The tree control (a), the attribute panel (b) and the toolbar (c) ............... 105  
7.3 (a) Statechart Plugin (b) Feature Plugin (c) Abstract Syntax Plugin ......... 106  
7.4 The containment-chain can differ according to the represented aspect ....... 107  
7.5 Main steps of concrete syntax definition in VMTS ................................... 109  
7.6 Graphical editor for Regions ................................................................. 109
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.7</td>
<td>The steps of compiling in VMTS OCL Compiler</td>
<td>110</td>
</tr>
<tr>
<td>7.8</td>
<td>The OCL Class Library</td>
<td>111</td>
</tr>
<tr>
<td>7.9</td>
<td>Typical Component Hierarchy in VMTS Parallel Transformation Engine</td>
<td>113</td>
</tr>
<tr>
<td>7.10</td>
<td>The Statechart Metamodel and an example model</td>
<td>115</td>
</tr>
<tr>
<td>7.11</td>
<td>UML models in VMTS</td>
<td>116</td>
</tr>
<tr>
<td>7.12</td>
<td>The Flowchart and the Nessie-Schneidermann created by VPD</td>
<td>117</td>
</tr>
<tr>
<td>7.13</td>
<td>Modeling GSM Networks</td>
<td>118</td>
</tr>
<tr>
<td>7.14</td>
<td>Concrete syntax of Protocol DSL</td>
<td>119</td>
</tr>
<tr>
<td>7.15</td>
<td>Case study Metamodel and Model</td>
<td>120</td>
</tr>
<tr>
<td>A.1</td>
<td>E-based composition</td>
<td>132</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Performance factors - Network overhead .................................. 99
6.2 Performance factors - complexity ratio .................................... 100
6.3 Performance factors - complexity ratio 2 .................................. 100
6.4 Performance factors - metaconflicts ....................................... 101

7.1 Performance of model transformation approaches ....................... 112
Abstract

The increasing popularity of modeling has attracted the attention to create, handle and transform customizable, but precise visual languages. This thesis addresses issues of customizable visual language definition and transformation. In order to improve flexibility, visual languages are defined by metamodels, while transformations are realized as model transformations. The novel results can be divided into three main parts.

The first contribution deals with the definition of visual languages. Firstly, the attributes of metamodeled visual languages are focused. A new attribute structure is provided defined by a formal syntax. The structure is highly flexible and it can be used on any metamodeling layer. An instantiation transformation is also elaborated for the attribute structure. Secondly, the appearance definition of visual languages is focused. A method is provided, which can model the appearance by using a special visual language and map appearance definitions to topological definitions by model transformation.

The second contribution deals with constraints of visual languages. In order to define precise visual languages, the language definition is often extended by constraints. The contribution extends one of the most popular constraint languages, OCL to support the n-layer attribute structure. Moreover, optimization methods are provided for OCL-based constraint handling. The aim of the algorithms is to reduce the number of model queries by relocating the constraint, or by applying a special caching method during the constraint evaluation.

The third contribution focuses on the transformation of visual languages. A new model transformation approach is presented that supports executing transformations in parallel. Firstly, the transformation-level parallelism is focused, where the transformation rules are handled as atoms. Then, the rule-level parallelism is introduced, which uses parallelism during the matching process. The parallelism approaches are unified and their performance is analyzed by measurements.

In order to illustrate the practical applications of the results, the Attribute panel, AGSI Compact Framework, VMST Presentation Framework, VMST Presentation DSL, OCL Compiler and the AGSI Parallel Transformation Engine components of the Visual Modeling and Transformation System have been developed. The application of the results includes efficient construction of visual languages, including e.g. UML diagrams, resource editors, graphical programming languages, and transformation control flow languages. The application of constraint optimization is transparently used in modeling, while the parallel model transformation engine is used in large, industrial models, where the sequential transformation would be unexpectedly slow.
Összefoglaló

A modellezés növekvő népszerűsége a testreszabható, de precíz vizuális nyelvek létrehozására, kezelésére és transzformációjára irányította a figyelmet. Az értekezés olyan konstrukciókat mutat be, amelyek a testreszabható modellek létrehozásához és feldolgozásához szükségesek. A vizuális nyelveket metamodellézéssel, míg a transzformációt modelltranszformációval adjuk meg. Az értekezés elméleti eredményeit három főbb témaoszlopra soroltam.


A második témaoszlopot a vizuális nyelvek kényezteivel kapcsolatos konstrukciókat mutatja be. A témaoszlop az egyik legnépszerűbb kényeztnyelvet, az OCL-t egészíti ki az n-szintű attribútumhierarchia támogatásával. A témaoszlop ismertet továbbá két OCL-alapú optimalizációs algoritmust is. Ezek az algoritmusok a modell lekérdezéseinek számát csökkentik a kényeztek áthelyezésével, ill. egy speciális gyorsítótár alkalmazásával.

A harmadik témaoszlopot a vizuális nyelvek feldolgozásával foglalkozik. Egy új modelltranszformáció megközelítés kerül bemutatásra, amely támogatja a transzformációk párhuzamos végrehajtását. Elsőként a transzformációszintű párhuzamosságot tárgyaljuk, ahol a transzformációk szabályokat atomként kezeljük. Másodikként a transzformációk szabályszintű párhuzamosítás kerül bemutatásra, ami a mintakeresés során alkalmaz párhuzamossáztást. A témaoszloport végén ismertetésre kerül a két párhuzamos megközelítés összeillesztésének módja, valamint a megoldások teljesítményével kapcsolatos méréseket és elemzéseket is bemutatjuk.

Preface

Dedication

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

Nyilatkozat¹

Alulírott Mezei Gergely 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.

Budapest, 2007 Október

(Mezei Gergely)

¹A bírálatok és a védésről készült jegyzőkönyv a későbbiekben a Dékáni Hivatalban elérhetőek.
Acknowledgments

First of all, I am really grateful to my family. I am very thankful to my parents for their altruist help even in the helpless hours. I would like to thank my wife, Anett for her support and for her infinite tolerance.

I am indebted to Tihamér Levendovszky, who inspired my research, and gave me countless ideas to continue my research, when I was stuck. I am very thankful to him for the many-many proofread papers, his advices and professional support. Without his help, this thesis could not have been written.

I am also indebted to Hassan Charaf for providing the financial and the human conditions of the work and his useful advice related to the research. I would like to thank to István Vajk his stimulating talks at the beginning of the fall semesters.

I am grateful to László Lengyel for the common work in the implementation, his patience in answering my questions and I would like to thank him his useful research related questions and remarks. I would like to thank the common work for László Angyal, Bence Kővári, Tamás Mészáros and Tamás Vajk as well. I would like to thank the colleges at the Department of Automation and Applied Informatics (Budapest University of Technology and Economics) their help.

Thanks to the reviewers of the papers their very useful advices, criticism, suggestions, remarks and questions. The reviews helped me a lot during the work.
Chapter 1

Introduction

Modeling plays an essential role in software engineering. It permits to raise the level of abstraction and it helps in defining steps of software lifecycle. Additionally, visual modeling languages have the advantage to illustrate the modeled problem in a simple yet expressive way. The high level of abstraction and the ability to describe problems in a graphical form have dramatically improved the achievable complexity of software in the last few decades [Jézéquel, 2003].

During the evolution of visual languages, many language families and language paradigms were created (e.g. [Jacobson, 1992a] [MAML Official Homepage, 2000] [Booch, 1993]). One of the most successful of these is the UML [UML 2.0, 2006]. UML is a generic modeling language family in the meaning that UML uses the same tool set regardless from the modeled domain. This generosity makes hard to use UML in case of special domains [Rumbaugh et al., 1999]. Domain-Specific Modeling Languages (DSML) [Domain-Specific Modeling, 2007] is a means to solve this issue and model each domain with its own, special topological rules and notation.

Metamodeling is an efficient way to support creating and editing DSMLs [Nordstrom, 1999]. In case of metamodeling, the modeling languages are defined by a special model, the metamodel. A metamodel acts as a set of topological rules for the model level: it defines the available model elements, their attributes and the possible connections between them. The metamodel-model relationship is referred to as instantiation. This instantiation relationship can be generalized to more than one modeling levels, for instance we can define the instantiation of the model, or the metamodel of the metamodel. As result, we obtain an n-layer modeling hierarchy. In order to support this n-layer hierarchy, there is a need for a n-layer instantiation method. Considering the n-layer instantiation, we have recognized two main issues to solve: the instantiation of topological rules and the instantiation of attributes. While the first issue has been solved [Levendovszky, 2006], the instantiation of attributes has not been standardized.

Besides the topological rules, it is also important to use the correct, domain-specific notation. However, usual metamodeling techniques cannot describe the refined appearance of model items. Several tools exist, which require the decoration of the metamodel by appearance-based properties (GME, [Lédeczi et al., 2001]), or which are based on customized source code (Eclipse GEF [Budinsky et al., 2003]).

Visual language definitions have the tendency to be imprecise. This applies to UML [Warmer and Kleppe, 2003] as well as metamodeled visual languages. Several approaches exist
to describe a precise modeling framework based on different mathematical representations e.g. [Varró and Pataricza, 2002] [Große-Rhode, 2004]. Although a complete precise metamodeling approach is hard to create, the impreciseness of visual language definition can be handled by attaching textual constraints to the visual languages. OCL [Warmer and Kleppe, 2003] is one of the most popular, formal constraint language. There exist several tools that support OCL constraints, however, constraint optimization is not supported by these tools.

Besides modeling, it is important to support model processing techniques as well to create a model-based software engineering approach [Metzger, 2005]. In general, model transformation techniques can be categorized as either model-to-model transformation or model-to-code translation [Czarnecki and Helsen, 2003]. In a typical model transformation framework, transformation rules and application rules are written in a special potentially visual language that can be either graphical [Agrawal et al., 2003] or textual [Gray et al., 2003].

Graph rewriting-based graph transformation is a mean to apply model transformation. This model transformation approach has a mathematical background and it is relatively simple to visualize the transformation rules. The main problem with the approach is that its performance does not scale up with the size of the underlying input model and with the complexity of rule definitions. The main reason for this is the fact that the transformations in this approach are based on isomorphic subgraph matching. There exists several results to accelerate matching [Czarnecki and Helsen, 1999] [Itai and Rodeh, 1978], but these approaches focus on special graphs. An efficient way to solve problems with large computational complexity is to solve them in parallel.

The goal of this thesis is to describe an approach that provides an efficient way to create, edit, maintain and transform visual languages by supporting the flexibility of n-layer metamodeling, the visualization capabilities of domain-specific languages and the creation of precise language definitions. In order to create an efficient solution, the thesis focuses on optimizing constraint handling and parallel application of model transformations as well.

1.1 Thesis Contributions

The thesis is organized around four concepts summarized in three topics that together forms a flexible, configurable and efficient solution to create and process metamodeled visual languages.

- The Definition of Visual Languages.
- Optimized, Metamodel-Based Constraint Validation.
- Truly Parallel Graph Transformation.

To illustrate the achieved contributions, several components namely the attribute translation, the VMTS Presentation DSL, the VMTS Presentation Framework, the OCL compiler, the AGSI Compact Framework and the VMTS Parallel Transformation Engine of a software framework called Visual Modeling and Transformation System (VMTS) have been developed.

In order to facilitate an efficient and flexible solution to create and maintain visual languages, several issues must be solved. Metamodeling is one of the most popular way to define visual
languages. N-layer metamodeling can increase the productivity and simplify the language definitions. However, there is a need to support an instantiation that supports topological rules (which issues have been solved already [Levendovszky, 2006]) and attribute handling as well. Furthermore, it is necessary to formalize the instantiation and the attribute structure in order to ensure the preciseness of the modeled language. It is also essential to provide an approach that is capable of defining the visualization of the visual language elements, since metamodels are invented for different aspects of language definition.

- A general attribute structure and an instantiation transformation are presented, which support layer transparency between the modeling layers.

- Both the attribute structure and the instantiation transformation are formalized. The definition of valid attribute structures is given.

- Propositions are proven that show that the n-ary composition of the instantiation transformation exists and produces valid models if the initial model is valid.

- The computational complexity of the transformation is analyzed.

- A new approach is presented, which allows to define the appearance of visual language elements and map this definition to the metamodel.

- It is shown that the appearance definition can be processed by model transformation and the transformation always terminates.

Textual constraints are often used to eliminate the incompleteness of visual languages stemmed from the visual definitions. OCL is one of the most popular constraint languages. OCL was created to extend UML definitions, but it is widely used in (meta)modeling as well. It is worth creating a dialect of OCL that supports n-layer metamodeling and the attribute structure discussed earlier. Moreover, the optimization of constraints would be useful, which is not supported in existing approaches. Both the new OCL dialect and the optimization algorithms are to be formalized in order to preserve the preciseness of OCL.

- An OCL dialect is presented that supports n-layer attribute structures.

- A new formalism, the OCLASM is provided for OCL. The formalism describes the new dialect and formalizes the original language as well.

- An algorithm is elaborated, which can reduce the number of model queries by relocating the constraints. Propositions are proven to show that the algorithm supports multiplicities in the model as well.

- An efficient model query optimizer algorithm is presented that is based on a caching mechanism.

- The algorithms are formalized and their correctness is proven formally.
The performance of graph rewriting-based model transformations is heavily affected by the size of the host model and the complexity of rewriting rules. Since the reasons can be traced back to an NP-complete problem (subgraph isomorphism), the performance cannot be improved by simplifying the problem in general. However, parallelism is a way to improve the performance. The parallel execution of rules (transformation-level parallelism) is useful in certain cases, while using parallelism in applying a single rewriting rule (rule-level parallelism) is also possible.

- A new, offline method is proposed that can find metamodel-based rules, which are always applicable in parallel. An extension is given for the method that can apply possibly conflicting rules in parallel. Heuristics are presented to reduce the probability of conflicts.

- The computational complexity of applying a single rewriting rule is analyzed. Algorithms are provided that support rule-level parallelism and reduce the time needed by calculations in the general case.

- A theoretical framework and distributed algorithms are presented that unify transformation-level and rule-level algorithms.

- The performance of the parallel execution is analyzed and compared to the performance of the sequential case.

As illustrated in this thesis, results related to the enlisted issues constitute an efficient solution to define, validate and process visual languages.

### 1.2 Thesis Structure

The rest of the thesis is organized as follows:

- Chapter 2 is devoted to illustrate the motivations of the results presented in the thesis. The chapter serves as a general introduction to the concerned topics, namely modeling (UML, Domain-Specific Modeling, metamodelling), constraint handling and model transformation.

- Chapter 3 presents the theoretical background of the thesis. A detailed introduction is given about Abstract State Machines (ASMs), which are used both in formalizing the attribute structure and OCL. The introduction contains a comparison of set theory and ASM as well that is referenced in formalizing OCL. The chapter elaborates the theoretical basis of graph transformations. Moreover, the specialities of model transformations used in our approach are discussed.

- In chapters 4, 5 and 6 the scientific results of the theses are contributed. These chapters introduce the background and motivation of the given topic, describe the related work, and discuss the contributions themselves. The results are divided into three parts: (i) visual language definition (Chapter 4), (ii) optimized, metamodel-based constraint validation (Chapter 5), and (iii) truly parallel graph transformation (Chapter 6).
Chapter 1. Introduction

- Chapter 7 introduces the components of the Visual Modeling and Transformation System (VMTS), which were created based on the presented theoretical results. The chapter discusses several minor, mainly practical results as well, such as dual-container handling, or the basis of a universal metamodeling framework. Furthermore, to illustrate the practical relevance, some of the most important application of the results is shown: (i) supporting UML, (ii) defining visual programming languages (e.g. Flowchart diagrams), (iii) using appearance definition in network modeling, (iv) constraint optimization case-study, and (v) several other case studies in a nutshell.

- Chapter 8 summarizes the thesis and outlines the future work.

- The Appendix contains some of the detailed proofs (Appendix A).
Chapter 2

Motivation

In software engineering, it is essential to have an adequate definition of the problem to solve. This goal can be achieved by providing a representation of the problem, which is self-explanatory and understandable for the project participants. An efficient and popular way is to use models, especially, visual models. Visual representation can simplify the problems by using a high abstraction level, and it can illustrate the solution in a graphical, easy-to-understand way. Having a visual modeling language that is (i) well-defined, (ii) unambiguous and at the same time (iii) that supports the domain-specific properties of the modeled problem set is a way to simplify software development.

As mentioned, visual languages have to be both unambiguous and flexible. There exists many approaches addressing this duality, but it is hard to support both requirements simultaneously. In general, the more well-defined a modeling language is, the less flexible it is. Metamodeling approaches usually offer a compromise, since they allow defining modeling languages and they often use formal basis as well. However, these approaches limit the number of modeling layers that raises the question, whether a metamodeling approach can be constructed that does not have this limit. Moreover, the flexibility of metamodeling is limited in creating visual languages: it is an open issue how metamodeling techniques could be extended to be able to define the appearance of the model items as well.

Although metamodeled visual languages are meant to be unambiguous, they are not always precise [Warmer and Kleppe, 2003]. In case of UML, similar problems led to create OCL and to attach OCL constraint to UML model items. It has already been shown that UML-based metamodeling and in more general, approaches which use limited number of metamodeling layers, can be extended similarly by OCL [Kolovos et al., 2006]. However, it is questioned whether OCL is extendable to support a real, n-layer attribute and modeling structure. Another issue – in connection with OCL – is to have an efficient constraint handling solution. Besides the implementation details, it would be useful to have a theoretical basis to optimize the execution of the evaluation of OCL constraints.

In order to respond the changing requirements and improve the level of reusability, there is a need to have model processing techniques. Obviously, efficiency is important in this case as well. The performances of model transformation engines are rather different, but most of the engines have difficulties with supporting large models. Distributed computing techniques
are used widely, wherever there is a need to improve the computation capabilities in a cost-efficient way. It is an open issue to find the conditions, the circumstances in which model transformations, especially the popular graph rewriting-based transformations are applicable in parallel. Moreover, the factors that affect the performance gained from the parallel execution must also be found.

This chapter reviews the industrial demands, theoretical and practical backgrounds which have led to the extensive research of visual language definition and processing. An overall view is provided here, the detailed description can be found in the closely related chapters. Firstly, the concepts of modeling and the role of constraints are introduced. Secondly, the concepts of model transformation are discussed in a nutshell, followed by the introduction to the most popular model-based software engineering methodologies. Finally, the open issues are concluded and the chapter summary is given.

2.1 Modeling

Nowadays, software engineering is almost unimaginable without modeling. Visual modeling languages have greatly increased the productivity. These modeling languages, language families and even the underlying basic concepts have evolved continuously in the last few decades. This section introduces some of the main milestones of this evolution.

The first structures used in software modeling were flowcharts and data flow diagrams. Later, these structures have been adopted by the structured software development paradigm and several new diagram types have been created (e.g. Jackson diagrams [Jackson, 1975]). The next step of the evolution came with the object-oriented paradigm, which has produced several different and sometimes contradictory models and notation systems e.g. [Rumbaugh et al., 1990] [Jacobson, 1992b] [Booch, 1993]. In contrast, Unified Modeling Language (UML) [UML 2.0, 2006] offered a universal solution to object-oriented software and hardware modeling.

2.1.1 The Unified Modeling Language

The Unified Modeling Language (UML) is one of the most well-known specification language for software and hardware modeling. UML is a general-purpose modeling language (more precisely a modeling language family) that defines the graphical notation used in the modeling as well. UML is defined at the Object Management Group [Object Management Group, 2007] by the UML metamodel, a Meta-Object Facility metamodel (MOF) [MOF Infrastructure, 2003]. UML supports specifying, visualizing, constructing, and documenting software and hardware systems. The evolution of modeling languages can be noticed in case of UML as well. Earlier versions (1.1, 1.5 [UML 1.5, 2004]) tried to satisfy the requirements of software industry, while the new major version, UML 2.0 [UML 2.0, 2006]) contains a completely new superstructure with much more flexibility. UML 2.0 added several new features such as hierarchical decomposition of structure and behavior, or layered architecture to facilitate the incremental implementation and compliance testing [Kobryn, 2004]. Moreover, as an extension and subset of UML, the Systems Modeling Language (SysML) [Sysml, 2007] has been created that is a Domain-Specific Modeling language for systems engineering.
Since the beginning, UML has played an essential role in evolution of model-driven technologies, such as Model Driven Development (MDD) [Atkinson and Kühne, 2003], or Model Driven Engineering (MDE) [Poole, 2001].

The main advantages of UML are the standard graphical notation and the well-accepted standardized topology definition of the modeling language. However, in the last few years, the inflexible, overly general model definition has attracted the attention to specialized modeling languages that support only one domain (or domain family), but offer more efficiency and a domain-specific tool set.

### 2.1.2 Domain-Specific Modeling

Domain-Specific Modeling (DSM) [Domain-Specific Modeling, 2007] is a modeling methodology to design and develop software and hardware systems. DSM is based on domain-specific modeling languages (DSMLs), which are mainly visual languages representing the specialties of the target domain. By supporting a domain-specific tool set and notation, these languages simplify specialized illustration of problems even for non-IT experts, which improves efficiency both in modeling and model-based approaches [Soley et al., 2004]. Moreover, in case of DSM, the abstraction level is not set by global rules as in UML. This means that the users can freely customize the abstraction level, which can produce more compact and more expressive model definitions. Domain-specific models are often reused in, or transformed to other models.

Besides the advantages of domain-specific modeling languages, it is usual problem to define these languages because of their flexibility. Similarly, it is problematic to create a tool that supports DSMs, because of the large differences in target domains. However, these issues can be solved by metamodeling.

### 2.1.3 Metamodeling

Metamodeling is an efficient technique to define visual languages including domain-specific languages. The concept of metamodeling has been used in describing syntactic rules of UML models as well. Simply said, a metamodel is the model of a model. Although metamodeling techniques have been discussed in many papers, articles and books [Atkinson and Kühne, 2001] [Atkinson and Kühne, 2002] [Sztipanovits and Karsai, 1997], as a matter of fact, the term metamodel is still a matter of discussion [Metamodel Homepage, 2007]. We use the following definitions for metamodel and metamodeling:

**Definition 2.1.** The metamodel is a paradigm, a set of rules that the modeling environment should keep during the modeling.

**Definition 2.2.** Metamodeling is a process describing the creation of the metamodel and the models instantiating the metamodel.

The presented definitions are the interpretation that the Object Management Group (OMG) uses when describing UML as a metamodel. Moreover, OMG has created Meta Modeling Facility (MMF) in order to create a precise definition of UML 2.0 [Reggio, 2002] [UML 2.0, 2006].
Nowadays, metamodeling plays an essential role in designing visual languages just like UML did it a few years ago. Metamodel defines the available object types at the model level, the attributes of the model items and the possible connections between the model items as well. These rules have to be automatically forced by the metamodeling framework. Nevertheless, metamodel supports not only the flexible creation of visual languages, but simplifies adopting to changes in the domain as well. Due to the latter feature, metamodeling is particularly popular in defining domain-specifying modeling families.

One of the main issues in connection with metamodeling is that it is not intended to describe the notation of the target domain, but only its topological definition. Although the topological information seems to be more important, the appearance of the model items is of great importance as well. The domain-specific notation makes it possible to explain the problem to domain experts without software engineering experiences. Without the customized notation, the model can offer an abstract view only that is much harder to understand. Nevertheless the importance of appearance definitions, it is an open issue how to define this information and how to map the visualization to the data, namely to the metamodel.

Once a metamodel is defined, instances of this metamodel, more especially models which conform to the metamodel rules can be created. This type-instance relation is not restricted to two modeling layers. For example, we need three modeling layers for the classical UML Class diagram – Object diagram models, because the rules of the class diagram have to be defined in a Meta class diagram. Although three, or four modeling layers are theoretically enough to describe practical problems of software engineering, increasing the number of modeling layers can simplify the modeling process. The more modeling layers we can use, the more accurate we can refine the abstraction represented by the models, i.e. the smaller the gap is between the abstraction level of the neighbor modeling layers. Limiting the number of modeling hierarchy to three or four layers (as MOF does) is similar as if we would limit the number of components in a component-based software. It is obvious that all the practical problems can be implemented by using four components, but it is much easier and much more comfortable to have n components, where n depends on the actual problem set. Existing modeling tools usually do not support handling more than four modeling layers. However, having a modeling approach that supports n-layer modeling hierarchies would be useful not only in theory, but in practice as well.

2.1.4 Instantiation

In an object-oriented language, when a class is instantiated, the created instance is referred to as an object. An object can hold a reference to another object, this relationship is called link. Moreover, the object can set the value of its attributes defined in the class. A similar construct is used, when defining a metamodel. The link is modeled as an association which connects types (classes), while the attribute structure is defined by using the attributes of the given type. To draw a parallel, metamodels are related to the instance model as the UML Class Diagram is related to the UML Object Diagram. Metamodeling is hard to use without a proper definition of instantiation and without the exact definition of how metamodel levels are related to each other.
The most used instantiation transformation is defined by MOF. However, as elaborated in [Kurtev and van den Berg, 2004] and in [Evans et al., 1999], MOF implicitly uses several different instantiation relationships. More precisely relationships between levels M3, M2 and M1 apply a different instantiation relationship, than those between M1 and M0. Moreover [Atkinson and Kühne, 2001] discusses the concept of deep instantiation, where "deep" means that a metamodel element can affect not only its immediate instance layer, but also other layers in the instantiation chain. Deep instantiation, however, is not handled by MOF.

The UML Class diagram – Object diagram models form a typical example why we need deep instantiation. We define the structure of class variables on the M2 level, in Metaclass diagram. Here, we describe that a class variable can have a type, name and a privacy information (e.g. protected). The instantiation of Metaclass diagram is the Class diagram (M1 level). Here, we define the variables themselves, e.g. that the class Processor has a variable CoreSpeed. When we instantiate the M1 level, we create Object diagrams (M0). In an object, we need to know the exact variable definition in order to check access rights (e.g. private variables cannot be accessed from outside the object). In order to obtain this information, we need to know not only the settings of the variable, but the variable structure as well. However, the structure has been defined in M1, not in M2. Obviously, we could hard-wire the variable structure in the modeling tool as well, but this would contradict to the metamodeling concept. Moreover, hard-wiring the structure would not help to support future domains.

The aforementioned instantiation techniques are based on the four levels of MOF, they do not support n-layer metamodeling. The definition of an instantiation method consists of at least two parts: the instantiation of topological rules (available model types and their connections) and the instantiation of attributes (properties of the model types). These issues must be solved in order to create a transformation that is capable of supporting n-layer metamodeling. The thesis [Levendovszky, 2006] has shown that the instantiation of topological rules can be described, but similar algorithms have not yet been presented to define the attribute instantiation.

If the number of modeling layers is limited, it is possible to create a separate instantiation algorithm for each neighbor layers. However, this is not possible in case of n-layer metamodeling, where the number of layers is not limited. Here, we need a layer transparent instantiation mechanism. Layer transparency means in this case that the same instantiation algorithm is used regardless on which modeling layer it is applied. Note that this uniformity can grant that the modeling hierarchy is not limited in depth.

2.1.5 Constraints

Visual languages have the tendency to create incomplete, informal, imprecise, and sometimes even inconsistent models. There is a need to describe additional constraints about the objects in the model. Practice has shown that using natural languages to describe these constraints results in ambiguities. Several approaches, several constraint languages have been created such as Catalysis [Catalysis, 2007] [D’Souza and Wills, 1998], Syntropy [Cook and Daniels, 1994] [Syntropy, 2006], or Object Constraint Language (OCL) [Warmer and Kleppe, 2003]. OCL is possibly the most popular, or at least the most widely used, constraint language.
Chapter 2. Motivation

OCL is a formal, textual and declarative language devoted to help the creation of precise models. Initially, OCL was only a formal specification language extension to UML. Later, it has been used with any MOF metamodel, including but not restricted to UML. Nowadays, OCL is used in a wide range of visual languages as a precise specification language. The textual constraints defined by OCL are attached to the model items of the selected modeling language. A constraint – in this case – means a restriction on one or more values of the modeled system. OCL supports querying the associations, or the attributes of the model item and offers various functions for basic types such as integer, Boolean or real. OCL is a key component of the new OMG standard recommendation for transforming models, namely the QVT specification [QVT, 2005].

In OCL, six types of constraints are distinguished: (i) Invariants define an expression that must be true for all instances of the selected type at any time. (ii) Definitions allow creating a derived attribute or operation. Each instance of the selected type has an attribute or operation that conforms to the given definition. Note that all operations defined in this way are considered query operations. (iii) A derivation rule specifies a value and a context element (attribute, or association end). It specifies that the value must always equal to the selected context element. (iv) An initial value is the value that the attribute or association end will have at the moment when the contextual instance is created. (v) A precondition of an operation of one of the model items is a restriction that must be satisfied just before the execution of the operation. Postconditions are similar constructs to define an expression that must be true after the operation has finished. (vi) The body of a query operation can be defined by the body keyword.

OCL is a quite flexible language. This flexibility is one of the main reasons of its popularity. For example, preciseness is important not only in modeling, but in model transformation specifications as well. Besides the topological definition of model transformation rules, additional constraints must be defined to ensure the correctness of the properties. The flexibility of OCL makes it capable of describing constraints in an n-layer metamodeling system. Obviously, this dialect of OCL must be precisely defined (in a formal way) to be able to keep the advantages of OCL. Besides the flexibility of OCL, the performance of constraint evaluation is also important due to the wide range of usage.

2.2 Model Transformation

In modeling, model processing techniques are essential, unless we use models only to documentation purposes. There are several ways to process models such as model traversing [Mezei et al., 2005e] or model transformation. Traversing the models is easier to implement, but visual model transformations are more expressive.

Model transformations are motivated by, for instance, OMG’s Model-Driven Architecture (MDA) [MDA, 2003] and Model-Integrated Computing (MIC) [Sprinkle, 2004]. The input of a model transformation is a model conforming to a given metamodel and the output of the transformations is another model conforming either to the same, or to a different metamodel. A model transformation may have several source models and several target models.

There exists many model transformation approaches. Some of these approaches are the following: (i) QVT [QVT, 2005] that is a model transformation standard created by OMG.
Chapter 2. Motivation

(ii) ATL [ATL Transformation language, 2007], which is a model transformation language developed to answer the QVT Request For Proposal. (iii) VIATRA [Csertan et al., 2002] that offers precise model transformations defined by formal syntax. (iv) GReAT [Agrawal, 2003], which is model transformation language for Model Integrated Computing (Section 2.3.2). (v) The AGG [Taentzer, 2004] graph transformation engine (vi) Tefkat [Tefkat Transformation language, 2007] a transformation language and a model transformation engine. (vii) Kermeta [Kermeta Transformation language, 2007] that is a programming and model transformation language.

As the wide range of transformation languages shows, it is hard to represent a model transformation in a rigorous, formal yet flexible and efficient way. The flexibility is granted in several approaches by allowing modeling the transformations themselves as a special, domain-specific visual language. Rigorous rule definitions are achieved in this case either by adding textual constraints as described in the previous section, or by formalizing the whole transformation approach as in [Varró, 2004]. Moreover, it is important to have an efficient model transformation engine, i.e. a transformation engine with good performance factors.

2.3 Model-Driven Engineering

Model-Driven Engineering (MDE, [Poole, 2001]) uses models as primary engineering artifacts in engineering lifecycle. MDE is not restricted to software modeling, it can also be applied to system and data engineering. MDE is a very large family of modeling and model transformation initiatives, for example Model-Driven Architecture (MDA), Model-Driven Development (MDD) or Model Integrated Computing (MIC).

2.3.1 Model-Driven Architecture

Model-Driven Architecture [Fowler and Scott, 2000] [Kleppe et al., 2003] [Mellor and Balcer, 2002] [Mellor et al., 2004] [MDA, 2003] supports model-driven engineering of software systems. MDA provides guidelines to structure specifications expressed as models. MDA separates design from architecture. More precisely, it defines a platform-independent model (PIM), a Platform Definition Model (PDM) and several platform-specific models (PSMs). PIM is usually created by using UML and other modeling languages capable of generating platform-dependent artifacts automatically by model compilers. MDA does not specify the model compiler to use, although the QVT [QVT, 2005] standard of OMG is a recommendation.

The main advantage of MDA is that it separates the universal designing concepts from the concrete, implementation-specific realization concepts. Using this dual abstraction level, MDA allows system developers to choose from the best and most fitting in both domains and greatly improves re-usability.

Since PIM, PDM and PSMs are often modeled by visual languages, MDA requires a modeling tool with support for creating and editing these types of languages. Another requirement of MDA is to have a model transformation engine capable of creating PSMs from PIMs.
Model Driven Development stresses the use of models in the software development life cycle and argues automation via model execution, model transformation and code generation techniques. Model Driven Development is a specific application of MDA to software development.

### 2.3.2 Model Integrated Computing

Model-Integrated Computing [Sztipanovits and Karsai, 1997] [Sztipanovits and Karsai, 2002] [Sztipanovits et al., 2003] is another model-based approach. MIC uses domain-specific models to synthesize applications. The key element is the extension of the role of models such that they form the “backbone” of a model-integrated system development process. In order to achieve this, MIC focuses on models, modeling environments and facilitates code generation from the models. Metamodeling environments and model interpreters together form the tool support for MIC.

MIC defines a development cycle as well. The first step of the cycle is to create a formal specification of the selected application domain. This specification is based on metamodeling techniques. The domain definition, i.e. the created metamodel is used to generate a domain-specific design environment (DSDE) by Meta-Level Translation. The generated environment can be used to create and edit domain-specific models. Then, an analysis is applied to the models. This is usually achieved by converting the models into another format such as executable code or configuration files for simulators. The conversion of the original model is called model transformation, it is performed by model transformers.

![Fig. 2.1. The basic concepts of MIC](image.jpg)

MDA and MIC has several similarities, more precisely, MIC can be considered as a special (domain-specific) MDA realization, where the focus is on developing the MDA process for specific domains.
2.4 Open Issues

Nowadays, visual languages lies at the heart of model-driven software engineering. The definition of the languages, the evaluation of constraints attached to language elements and the model transformation techniques processing the models are crucial in creating a model-driven approach.

The main topic of the presented research is the transformation-based support for visual languages. This topic, this aim can be achieved by solving several problems, weaknesses and shortcomings of the existing approaches. N-layer metamodeling needs a layer transparent attribute structure and instantiation. Metamodeling requires a method to describe the notation of visual language elements. Moreover, an OCL dialect is needed to support n-layer metamodeled attributes. In order to improve the performance of constraint validation, constraint optimization algorithms should be created. Finally, parallel model transformations are required to create an efficient model transformation approach.

The open issues related to the efficient model transformation are as follows.

- **N-layer attribute structure.** To support n-layer metamodeling, an attribute structure should be created. This structure must be layer transparent and flexible enough to describe the attributes of domain-specific languages. The attribute structure must be formalized. A precisely defined, layer-transparent instantiation transformation is also needed.

- **Supporting appearance definitions.** Metamodeling does not support defining the appearance of visual models. A new method is needed that fits the metamodeling concepts and is able to describe the notation of the language. A model processing technique is also needed to map and bind appearance definitions to the topological definitions.

- **Constraints extensions.** In order to make constraint validation possible in n-layer metamodeling environments, an extension, a new dialect must be provided to one of the existing constraint languages. The new dialect should be formalized.

- **Optimized constraint evaluation.** There is a need for an efficient constraint handling solution. To achieve this, the optimization of OCL constraints must be supported. Moreover, the correctness of the optimization algorithms must be proved in a formal way.

- **Parallel model transformations.** The performance of graph rewriting-based model transformations must be improved by parallel execution of the transformations. An examination is needed to explore the parallelization possibilities at transformation level and at the level of rewriting rules. Distributed algorithms must be created and the efficiency of these algorithms must be proven by performance measurements.

2.5 Visual Modeling and Transformation System – Overview

Visual Modeling and Transformation System (VMTS) [VMTS Homepage, 2007] [Lengyel, 2006] [Levendovszky, 2006] is a metamodeling environment and model transformation framework. VMTS supports n-layer metamodeling with full transparency [Levendovszky et al., 2005] between the layers. In VMTS, metamodel rules are automatically forced when editing models. Moreover,
Chapter 2. Motivation

VMTS supports OCL-based textual constraints, which can be attached to metamodel items. Models and transformation rules are formalized as directed, labeled graphs. VMTS uses a simplified class diagram as its root metamodel (“visual vocabulary”).

In VMTS, model transformation is based on graph rewriting techniques. A domain-specific control flow language is used to specify the steps of transformations and rewriting rules are also specified by a visual language. During the model transformation process, the tool facilitates the validation of the constraints specified in the transformation rules.

The theoretical results discussed in this thesis have been validated in VMTS as a proof-of-concept implementation (Section 7.1).

2.6 Chapter Summary

This chapter has introduced some of the most important steps of the evolution of model-based software engineering. It has been shown that visual languages and model transformations play an essential role in the software development process nowadays. Flexibility and efficiency are two key features, but it is hard to ensure both of them simultaneously.

The chapter has followed the evolution of modeling approaches and concluded that the flexible definition of visual languages, especially domain-specific visual languages, is essential. Although the existing metamodeling approaches can describe several aspects of language creation, they have imperfections and shortcomings as well. One of the most important issues is how to create a generalized, layer transparent instantiation transformation in order to support n-layer metamodeling. Although the topological aspects of the issue have been solved already, the attribute handling does not have such a solution.

Another issue in creating visual languages by metamodeling is to model the appearance definition of the selected language. Metamodeling is not intended to contain this definition, thus, a new solution is required to extend the metamodel definitions.

Metamodeled and more generally visual languages have the tendency to be incomplete. This problem can be solved by attaching textual constraints to visual model definitions. OCL is possibly the most well-known constraint language and it would be an obvious solution to use OCL in metamodeling. However, a new dialect is required that supports the n-layer metamodeling constructs, especially the n-layer attribute structure. Moreover, it is important to have an efficient constraint evaluation method. Existing approaches do not support constraint optimization though, thus, it is an open issue to create algorithms that would accelerate the constraint evaluation.

Besides the definition of visual languages, it is also important to be able to transform them to other models, source code, or to a customized model representation. Graph transformation is a popular method for transforming models, but the performance of the graph transformation approaches does not scale up with the size of the host model. Parallel execution of model transformation would mean a solution to this issue, but it is an open question, how to find rules that are applicable in parallel automatically and how to apply parallelism in an efficient way.
Chapter 3

Theoretical Background

This chapter introduces the theoretical background – constructions, techniques, formalisms and results – of the thesis. The chapter is intended to give an overview that can act as the basis of the future chapters. Firstly, Abstract State Machines are introduced. This is a popular formalism technique, which will be used in formalizing the attribute structure (Chapter 4) and in the new formalization of OCL (Chapter 5). Then, the basic constructs of category theory are presented. Category theory is important, because the graph transformation approach used in Chapter 4 and Chapter 6 is based on categorical constructs. In Section 3.3 we present this graph transformation approach in detail, including the most related results produced by the approach. The specialities of our transformation approach are elaborated as well. Finally, a summary is given on the discussed technologies and approaches.

3.1 Abstract State Machines

Abstract State Machines (ASMs) – formerly called as evolving algebras – have been introduced by Yuri Gurevich [Böerger and Stärk, 2003] as an attempt to bridge the gap between formal models of computation and practical specification methods. The main idea behind ASMs is that any algorithm can be modeled at its natural abstraction level by an appropriate evolving algebra. Built upon this idea, a methodology based upon mathematics was defined. The result is a simple methodology for describing simple abstract machines which correspond to the algorithms. According to [Abstract State Machines, 2007] ASMs have been used in many applications such as formalizing model transformations [Varró, 2004], the semantics of the C++ language [Wallace, 1995], the Java Virtual machine [Stärk et al., 2001], or the UML Statechart diagram [Compton et al., 2000].

3.1.1 Introduction to ASM

ASMs are finite sets of transition rules of the form

if Condition then Updates,
which transform abstract states, where \textit{Condition} (referred to as guard) under which a rule is applied is an arbitrary predicate logic formula without free variables. The formula of \textit{Condition} evaluates to \textit{true} or \textit{false}. \textit{Updates} is a finite set of assignments in the form of

\[ f(t_1, \ldots, t_n) := t, \]

whose execution is understood as changing (or defining, if there was not defined before) the value of the occurring \textit{function} \( f \) at the given arguments. Each function of an ASM has a fixed arity, which is a non-negative integer. Nullary functions are called \textit{constants}, however they can change their values, thus, they are handled more similar to variables of a programming language. By definition, all ASMs contain certain nullary function names (\textit{true}, \textit{false}, \textit{undef}), as well as the names of the usual Boolean operations and the equality sign.

The notion of \textit{ASM states} is the classical notion of mathematical structures where data come as abstract objects, i.e., as elements of sets (domains, universes, one for each category of data) which are equipped with basic operations (partial functions) and predicates (attributes or relations). Without loss of generality one can treat predicates as characteristic functions.

The notion of an \textit{ASM run} is the classical notion of computation in transition systems. An ASM computation step in a given state consists of executing simultaneously all updates of all transition rules whose guard is \textit{true} in the state, if these updates are consistent. Therefore, transition rules describe how states of an ASM changes over time. A set of updates is called \textit{consistent} if it contains no pair of updates with the same location, but with different results. For the evaluation of terms and formulas in an ASM state, the standard interpretation of function symbols is used.

Simultaneous execution provides a convenient way to abstract from irrelevant sequentiality and to make use of synchronous parallelism. This mechanism is enhanced by the following concise notation for the simultaneous execution of an ASM transition rule \( R \) for each \( x \) satisfying a given condition \( \varphi \):

\[ \text{forall } x \text{ with } \varphi \ R, \]

where \( \varphi \) is a Boolean-valued expression and \( R \) is a rule. We freely use abbreviations, such as \textit{where}, \textit{let}, \textit{if then else}, \textit{case} and similar standard notations which are easily reducible to the basic definitions above [Böerger and Stärk, 2003].

A priori no restriction is imposed either on the abstraction level or on the complexity or on the means of the function definitions used to compute the arguments and the new value denoted by \( t_i, t \) in function updates. The major distinction made in this connection for a given ASM \( M \) is that between \textit{static} functions which never change during any run of \( M \) and \textit{dynamic} ones which typically do change as a consequence of updates by \( M \) or by the environment. The dynamic functions are further divided into four subclasses. \textit{Controlled} functions are dynamic functions which can directly be updated by and only by the rules of \( M \). \textit{Monitored} functions are dynamic functions which can directly be updated by and only by the environment. \textit{Interaction} or \textit{shared} functions are dynamic functions which can directly updated by rules of \( M \) and by the environment. \textit{Derived} functions are functions which cannot be directly updated either by \( M \) or by the environment, they “inherit” the properties of the functions they are defined with.
3.1.2 Mathematical Definition of ASM

In an ASM state, data is available as abstract elements of domains which are equipped with basic operations represented by functions. Without loss of generality, relations are treated as Boolean-valued functions and view domains as characteristic functions, defined on the superuniverse which represents the union of all domains.

**Definition 3.1.** A vocabulary (signature) $\Sigma$ is a finite collection of function names. Each function name $f$ has an *arity*, which is a non-negative integer. The arity of a function name is the number of arguments that the function takes. Function names can be static or dynamic. Nullary function names are often called constants; but the interpretation of dynamic nullary functions can change from one state to the next, such that they correspond to the variables of programming. Every ASM vocabulary is assumed to contain the static constants $\text{undef}$, $\text{True}$ and $\text{False}$.

**Example 3.2.** The signature $\Sigma_{\text{bool}}$ of Boolean algebras contains two constants 0 and 1, a unary function name '-' and two binary function names '+' and '*'.

**Definition 3.3.** A state $A$ of the vocabulary $\Sigma$ is a non-empty set $X$, together with interpretations of the function names of $\Sigma$, where $X$ means the superuniverse of $A$. If $f$ is an $n$-ary function name of $\Sigma$, then its interpretation $f^A$ is a function from $X^n$ into $X$; if $c$ is a constant of $\Sigma$, then its interpretation $c^A$ is an element of $X$. The superuniverse $X$ of the state $A$ is denoted by $|A|$.

The *elements* of the state are the elements of the superuniverse of the state, and – according to the definition – the parameters of the functions are also elements of the superuniverse. If, for example, an ASM formalizes a graph rewriting technique, then the superuniverse is the set of IDs of all potential model elements as used in [Varró, 2004].

**Definition 3.4.** A sequential abstract state machine $M$ consists of a vocabulary $\Sigma$, an initial state $A$ for $\Sigma$, a rule definition for each rule name, and a distinguished rule name called the main rule name of the machine.

More information on the mathematical representation and the notation of ASMs can be found in [Böerger and Stärk, 2003].

3.2 Category Theory

Category theory [Pierce, 1991] [Barr and Wells, 1990] [Sabetzadeh and Easterbrook, 2003] [Segura ML, 2001] is a relatively young branch of algebraic topology. The main advantage of category theory is that it produces compact descriptions of the problems without losing preciseness [Ehrig, 1979b]. Category theory focuses mainly on functions, their domains and ranges are of less importance. The basic constructs used in category theory 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.
Definition 3.5 (Category). A category $C$ comprises:

1. **Objects**

2. **Arrows** (morphisms)

3. **Operations** assigning to each arrow $f$ an object, an object $\text{dom} f$, its domain, and an object $\text{cod} f$, its codomain (denoted with $f : A \rightarrow B$, if $\text{dom} f : A$ and $\text{cod} f : B$). The collection of all arrows with domain $A$ and codomain $B$ is denoted with $C(A,B)$.

4. A binary **composition operator** assigning a composite arrow $g \circ f : \text{dom} f \rightarrow \text{cod} g$ to each pair of arrows $f$ and $g$ with $\text{cod} f = \text{dom} g$, satisfying the following associative law: for any arrows $f : A \rightarrow B$, $g : B \rightarrow C$ and $h : C \rightarrow D$(with $A$, $B$, $C$, $D$ not necessarily distinct), $h \circ (g \circ f) = (h \circ g) \circ f$.

5. For each object $A$, there exists an (identity) arrow $\text{id}_A : A \rightarrow A$, satisfying the following identity law: for any arrow $f : A \rightarrow B$, $\text{id}_B \circ f = f$, and $f \circ \text{id}_A = f$.

Definition 3.6 (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 3.7 (Pushout, pushout complement). Let $C$ be a category, with two given arrows $b : A \rightarrow B$, $c : A \rightarrow C$. The triple is called the pushout of if the following conditions hold:

(i) $g \circ b = f \circ c$ (commutative property)

(ii) 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).

The triple $(C, c : A \rightarrow C, f : C \rightarrow D)$ is called the pushout complement of $(b, g)$, while $D$ is called the pushout object of $(b, c)$.

Fig. 3.1 shows the diagram representation of pushout. The pushout object (D) emerges from gluing two objects (B and C) along a common subobject (A).

![Fig. 3.1. A diagram representation of pushout](image-url)
### 3.3 Algebraic Graph Transformation Approaches

The algebraic graph transformation approach has been invented to generalize Chomsky grammars from strings to graphs [Ehrig et al., 1973]. The main idea behind the approach is to use the similarities of string concatenation and gluing construction of graphs. Based on these similarities, graph rewriting steps can be formulated by one or two gluing constructions. The transformation approach [Ehrig, 1979a] [Ehrig, 1986] is referred to as algebraic, because the underlying graphs are handled as algebraic constructions and gluing is defined by pushout in the category of graphs and total graph morphisms. There are two main branches of algebraic graph rewriting, namely the double pushout (DPO) [Rozenberg, 1997] and the single pushout (SPO) [Ehrig et al., 1999b] approaches.

In these approaches, the graph representation is based on the following definition:

**Definition 3.8. (Labeled directed graphs (LDG) [Rozenberg, 1997])** 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: $G = \langle G_V, G_E, s^G, t^G, l^G, l^E \rangle$, 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 the source and the target vertex, respectively; and finally, $l^G: G_V \rightarrow \Omega_V$ and $l^E: G_E \rightarrow \Omega_E$, which assign a label to a vertex and an edge from the appropriate alphabet.

Note that $s^G(X)$ is often denoted as $\text{src}(X)$, while $t^G(X)$ is denoted as $\text{target}(X)$, where $G$ is unambiguously determined by the textual context. Using the definition of LDGs, we can create the category of Graph, where the objects are labeled directed graphs (Definition 3.8) and the arrows are graph homomorphisms.

Graph transformations consist of rewriting rules, also referred to as productions containing a left-hand side (LHS) and a right-hand side (RHS) graphs.

**Definition 3.9. (Production)** A production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of finite graphs $L$, $K$ and $R$, called left-hand side ($L$), gluing graph ($K$) and right-hand side ($R$) respectively, and two injective graph morphisms $l$ and $r$.

When applying the production, the pattern defined in LHS is matched against the host graph forming a match, also referred to as the redex of the production rule. Then, this pattern is replaced by the pattern defined in the RHS. There are two well-accepted ways to replace the matched pattern with the pattern defined in the RHS. The first (referred to as gluing approach) applies the replacement by gluing RHS model items to the common vertices. The other approach (connecting approach) adds edges to the disjoint union of the context graph and RHS. To specify the placement of these additional edges, embedding rules are accompanied with the rewriting rule [Janssens and Rozenberg, 1980]. This second approach is not as popular as the first one, because it cannot specify rules like containing all outgoing edges of a node.

Using category theory in describing graph transformation results in a compact yet precise definition of the approach. Moreover, most of the results can be easily used in extensions of the original underlying structures (e.g. High-Level Replacement Systems [Ehrig et al., 1994]).
3.3.1 The DPO Approach

The name of the DPO approach refers to the fact that it describes rewriting rules by using two pushouts. In this approach, the application of the production is defined as follows [Ehrig et al., 2005a]: Given a production $p = (L \leftarrow K \rightarrow R)$ and a context graph $D$ that includes the interface $K$. The input graph $G$ of a graph transformation $G \Rightarrow H$ via $p$ is given by the gluing of $L$ and $D$ via $K$, written $G = L +_K D$. The target graph $H$ is given by the gluing of $R$ and $D$ via $K$, written $H = R +_K D$. We use graph morphisms $K \rightarrow L$, $K \rightarrow R$ and $K \rightarrow D$ to express how $K$ is included in $L$, $R$ and $D$ respectively. This allows defining the gluing constructions $G = L +_K D$ and $H = R +_K D$ as pushout constructions leading to the double pushout shown in Figure 3.2. The resulting graph morphism $R \rightarrow H$ is called the comatch of graph transformation $G \Rightarrow H$.

The context graph $D$ is obtained from the host graph $G$ by deleting all elements, which have pre-image in $L$ but not in $K$. Then, items are added (glued), which are included in $R$ without having pre-image in $K$. In the DPO approach, the match $m$ must satisfy the gluing condition consisting two parts: The identification condition states that different vertices in the rewriting rule cannot match the same vertex in the host graph on deletion. The dangling edge condition requires that if the rule specifies the deletion of a vertex, then it must delete all edges connected to the vertex as well. Related to the DPO approach, tutorials can be found in [Ehrig, 1979b] [Ehrig, 1986] [Ehrig et al., 1991] [Corradini et al., 1993] and a more complete summary in [Ehrig et al., 1999a] [Rozenberg, 1997].

Based on the DPO approach, several results that are related to the presented research have been created. The following subsections introduce a few from these results.

3.3.2 Parallelism Theorem

We can distinguish two different kinds of independency between the rules. Two rules are parallel independent, if each of them can still be applied after the other one has been finished. Two rules are sequential independent if they can be swapped.

Definition 3.10. (Parallel independence) Let $G \xrightarrow{p_1,m_1} H_1$ and $G \xrightarrow{p_2,m_2} H_2$ be two direct derivations from the same graph $G$ (Fig 3.3). The derivations are parallel independent, if $m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$. Or in categorical terms: There exists two graph morphisms $L_1 \xrightarrow{k_1} D_2$ and $L_2 \xrightarrow{k_2} D_1$ such that $l_2^* \circ k_2 = m_1$ and $l_1^* \circ k_1 = m_2$.

The intuitive meaning of the definition is that neither derivation rules can delete any elements, which is needed by the other rule.
Definition 3.11. (Sequential independence) Two consecutive direct derivations (Fig 3.4) \( G \xrightarrow{p_1,m_1} H_1 \xrightarrow{p_2,m_2} X \) are sequential independent, if and only if \( m_1^*(R_1) \cap m_2^*(L_2) \subseteq m_1^*(r_1(K_1)) \cap m_2^*(l_2(K_2)) \). In categorical terms: there exists 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 \( r_1^* \circ k_1 = m_2^* \).

The intuitive meaning of the definition is that \( p_2 \) cannot delete anything that has been explicitly preserved by \( p_1 \), and it cannot use any element generated by \( p_1 \).

In case of DPO approach, it is true that two arbitrary rules are parallel independent if and only if they are sequential independent as well. This equivalence is proved by the Parallelism theorem. Moreover, the work [Levendovszky, 2006] has shown that the equivalence holds for metamodel-based rewriting rules as well.
3.3.3 Concurrency Theorem

The Concurrency Theorem [Ehrig et al., 2005a] handles graph transformations that may be sequentially dependent. In general, we cannot combine subsequent direct graph transformations if they are sequentially dependent. However, it is possible to apply the productions concurrently by using an E-concurrent production $p_1 *_E p_2$.

**Definition 3.12. E-dependency relation** Given two productions $p_1 = (L_1 \xleftarrow{k_1} K_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xleftarrow{k_2} K_2 \xrightarrow{r_2} R_2)$, an E-dependency relation $(E, e_1, e_2)$ is given by a graph $E$ and injective morphisms $e_1 : R_1 \rightarrow E$, $e_2 : L_2 \rightarrow E$, which are jointly surjective.

**Definition 3.13. E-dependency production** Given two productions $p_1 = (L_1 \xleftarrow{k_1} K_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xleftarrow{k_2} K_2 \xrightarrow{r_2} R_2)$, the E-concurrent production $p_1 *_E p_2$ is a production $p = (L \xleftarrow{k} K \xrightarrow{r} R)$ computed based on the following diagram, where double squares (1)(2) and (3)(4) form double pushouts, and (5) is a pullback. Note that the injectivity of $e_1$ and $e_2$ implies that of $k_1, m_1, k_2,$ and $n_2$.

![Fig. 3.5. E-dependency production](image)

This definition can be applied recursively, using an E-concurrent production for $p_1$.

**Definition 3.14. E-related transformation** A transformation $G \xrightarrow{p_1,m_1} H \xrightarrow{p_2,m_2} G'$ is called E-related to $p_1 *_E p_2$ if there exist morphisms $h : E \rightarrow H_1$, $c_1 : K_1' \rightarrow D_1$, and $c_2 : K_2' \rightarrow D_2$ such that $h \circ e_1 = n_1$, $h \circ e_2 = m_2$, (6) and (7) commute and (8) and (9) are pushouts.

![Fig. 3.6. E-related transformation](image)
Using the definition above, an algorithm can be constructed to compute $E$-concurrent productions.

**Algorithm 3.1** Rule composition

1. Find an $E$-dependency relation $(E, e_1, e_2)$.
2. Compute the pushout complement $K'_1$ that makes (2) a pushout, otherwise go to Step 1.
3. Compute the pushout object $L$ that makes (1) a pushout.
4. Compute the pushout complement $K'_2$ that makes (3) a pushout, otherwise go to Step 1.
5. Compute the pushout object $R$ that makes (4) a pushout.
6. Compute the pullback object $K$ that makes (5) a pullback.

**Proposition 3.15** (Concurrency Theorem). Let $(E, e_1, e_2)$ be an $E$-dependency relation for the productions $p_1$ and $p_2$ leading to the $E$-concurrent production $p_1 \ast E p_2$.

1. **Synthesis**: Given an $E$-related transformation sequence $G \Rightarrow H \Rightarrow G'$ via $p_1$ and $p_2$, then there is a synthesis construction leading to a direct transformation $G \Rightarrow G'$ via $p_1 \ast E p_2$.
2. **Analysis**: Given a direct transformation $G \Rightarrow G'$ via $p_1 \ast E p_2$, then there is an analysis construction leading to an $E$-related transformation $G \Rightarrow H \Rightarrow G'$ via $p_1$ and $p_2$.
3. **Bijective correspondence**: The synthesis and analysis constructions are inverse to each other up to isomorphism.

![Fig. 3.7. Concurrency Theorem](image)

### 3.3.4 Embedding and Amalgamation

Amalgamated rules describe the synchronized application of two derivations. This is achieved by identifying a common subproduction. The subproduction specifies the shared effect of the original production rules. Given two rules $p_1$ and $p_2$, their synchronized production is denoted by $p_1 \overset{g_1}{\rightarrow} p_0 \overset{g_2}{\rightarrow} p_2$. The productions may be glued together by using $p_0$ leading to the amalgamated production $p_1 \oplus p_0 \oplus p_2$. A direct derivation $G \overset{p_1 \oplus p_0 \oplus p_2}{\rightarrow} H$ is called amalgamated derivation. Note that this derivation means the simultaneous application of $p_1$ and $p_2$, where the common actions ($p_0$) are applied only once. Therefore, if $p_0$ is empty, we receive the parallel production $p_1 + p_2$. The concept of this synchronization was firstly introduced in [Boehm et al., 1987].

Similarly to amalgamated productions, a graph can be also partitioned by using a common subgraph. A distributed graph $DG = (G_1 \overset{g_1}{\rightarrow} G_0 \overset{g_2}{\rightarrow} G_2)$ consists of two local graphs $G_1$ and
Chapter 3. Theoretical Background

25

$G_2$ sharing a common interface graph $G_0$ embedded into $G_1$ and $G_2$ by the graph morphisms $g_1$ and $g_2$. Gluing $G_1$ and $G_2$ yields the global graph $\oplus DG = G_1 \oplus G_0 G_2$ of $DG$. $DG$ is often referred to as a splitting of $\oplus DG$.

Is it possible to transform distributed graph by synchronized productions. Given a splitting of a graph $DG$, we can construct distributed derivations, which describe the transformation of a splitting of $DG$. Since synchronized rules can be amalgamated and distributed graphs may be glued together, it is a natural idea to describe a distributed derivation by an amalgamated one. By generalization of this idea we can obtain a distributed model transformation approach, which is based on the splitting of $DG$. This idea has been introduced in [Ehrig, 1987].

3.3.5 Termination Criteria

Often it is essential to decide whether a transformation will terminate or not, although this question is undecidable in general [Plump, 1998]. The paper [Levendovszky et al., 2006] elaborates an approach that simplifies to make this decision. The definition for the termination of a graph transformation system is taken from [Plump, 1998].

Definition 3.16. A graph transformation system $GTS = (P)$ terminates if there is no infinite sequence of direct graph transformations $G_0 \Rightarrow G_1 \Rightarrow ...$ applying rules from $P$ starting from any input graph $G_0$.

Using this definition an the previously introduced E-dependency relations, we can simplify the examination of termination properties.

Definition 3.17. Consider a possibly infinite sequence of graph productions $p_i$ ($i = 1, 2, ...$) and a sequence of E-dependency relations $((E_i, e_i^*, e_{i+1}^*))$ leading to a sequence of their E-based compositions ($p_i^* = (L_i^* \leftarrow K_i^* \rightarrow R_i^*)$) with $p_1^* = p_1$ and $p_n^* = (p_1 * E_1, p_2) * E_2 ... * E_n, p_n$.

A cumulative LHS series of this sequence is the graph series $L_n^*$ consisting of the left-hand side graphs of $p_n^*$. Moreover, a cumulative size series of a production sequence is the nonnegative integer series $|L_n^*|$.

Using the presented definitions, the following theorem holds:

Theorem 1. A $GTS = (P)$ terminates if for all infinite cumulative LHS sequences $(L_i^*)$ of the graph productions created from the members of $P$, it holds that

$$\lim_{i \to \infty} |L_i^*| = \infty.$$ 

Note that we assume finite input graphs and injective matches.

Using this theorem, it is able to examine termination properties of an arbitrary model transformation in a relatively simple way. We use this approach in examining the model transformation, which maps concrete syntax definitions to topological definitions of the selected metamodel.
3.3.6 Model Transformations in VMTS

The theoretical and practical results presented in the thesis have been implemented in the Visual Modeling and Model Transformation (VMTS) tool as mentioned earlier. This tool uses a DPO-based model transformation approach, but it also uses special constructs both in model transformation specification and in graph-rewriting. This section elaborates these constructs.

3.3.6.1 Transformation Control Flow

In VMTS, graph transformations are specified by a transformation control flow using the VMTS Control Flow Language (VCFL, [Lengyel et al., 2006e]). Basically, VCFL is a domain-specific visual language, more properly a stereotyped activity diagram (Fig. 3.8).

![Fig. 3.8. Example control flow using VCFL](image)

Model items of VCFL models are rewriting rules, or controller type items. At the moment, VCFL supports start and end node, decisions, fork and join operations. VCFL makes possible to define a strict execution order between the rules, unless parallel execution is explicitly set by a fork item. The rewriting rules, i.e. the rule type model items in the control flow, support the application of a rule as long as the matching of the rule is successful in the host model. These rules are referred to as exhaustive rules. VMTS supports external causalities [Mezei et al., 2005a] as well. External causality is a parameter passing solution between the rewriting rules. This feature allows to pass matched nodes of a rule to an other node, thus, the cost of re-matching can be reduced.

3.3.6.2 Specialities in Rewriting

VMTS uses metamodel-based rewriting rules. This means that LHS and RHS of the transformation rules are built from metamodel elements [Levendovszky et al., 2005]. This idea is taken originally from [Levendovszky et al., 2002]. The approach means that instead of a subgraph isomorphic to LHS, an instantiation of LHS must be found in the input graph. During the transformation process, a part of the input model must be found and that part must instantiate LHS, and the generated model is an instance model of RHS. To distinguish between the rule definition and its instantiation, we refer to as metamodel-based and instance model-based rules. The aforementioned Parallelism theorem cannot be applied automatically in case of metamodel-based rewriting rules. However, in [Levendovszky, 2006] it is proven that parallel and sequential independence is equivalent in this case as well.
The constructs that form a metamodel-based LHS specification are inheritance and multiplicity support [Levendovszky and Charaf, 2005]. Inheritance support is analogous to the type compatibility of object-oriented languages. The inherited class can be used where the parent class is expected. This means that an LHS element always matches its inherited types in the input model. This facilitates generalization in the transformation rules as well as abstract types. Multiplicity support is achieved by allowing multiplicity values on the association ends. 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 that does not contradict any other part of the match.

To simplify the construction of rewriting rules, we use a special domain-specific language. Models of this language allow setting the items of LHS and RHS in a visual form. However, the visual definition of the LHS - RHS relation is not always enough. Because of the multiplicity support and metamodel-based definition, a single item of the RHS can refer to more than one concrete model item in the host model. It is hard to describe the exact behavior of the model items visually in this case. To solve this issue, we allow specifying the relations between LHS and RHS by using a textual language. Originally, XML-based expressions were used, but the transformation engine now supports Imperative OCL [Vajk and Levendovszky, 2006] expressions as well. Imperative OCL is an extension of the OCL language [Warmer and Kleppe, 2003], which offers a wider range of traditional programming structures (e.g., foreach), than the original language and that can modify the model as well, in contrast to OCL that is intended to query the models only.

### 3.4 Chapter Summary

This chapter has introduced the theoretical background of the research including the formalization techniques used in Chapter 4 and in Chapter 5 and the algebraic background for metamodel-based model transformation.

Firstly, Abstract State Machines have been introduced. This relatively new formalization technique allows using an arbitrary abstraction level in defining the problem set and also supports hierarchical decomposition of the formal definitions. As the result of these features, ASM produces compact yet precise definitions of problems. ASM is particular efficient in formalizing pseudo-code like algorithms, such as the optimization methods of OCLASM presented in Chapter 5.

Category theory is another formalization technique that is used in the thesis mainly for describing graph transformations. Our graph transformation approach is based on the DPO approach that is also based on categorical terms.

The algebraic background of transformations has been introduced in Section 3.3. The definition of labeled directed graphs has been given, and the constructs of the DPO approach has been introduced. Based on the DPO approach, several result have been created, the chapter elaborated the results that are the most relevant to this thesis. The definition of sequential, and parallel independent rules and the Parallelism theorem are the basis of the parallelization algorithms presented in Chapter 6. The embedding and amalgamation of rules are important because they are used by other approaches aiming to create parallel transformations. The con-
currency theorem and the termination properties of the transformation are used in proving that the model transformation presented in Chapter 4 always terminates.

Finally, the specialties of our model transformation approach has been elaborated, namely the metamodel-based rewriting rules, the properties of the visual language to describe the transformation control flow, and the Imperative OCL-based rewriting mechanism.
Chapter 4

Defining Visual Languages

Metamodeling is an efficient method to create and handle visual languages. A metamodel acts as a set of rules for its model: it specifies the modeled types, the attributes of the types and the possible relations between them. The relationship between a metamodel and its model is called instantiation. Considering the instantiation, there are two issues: instantiation of topological rules (e.g. the instantiation of the association in the UML class diagram [UML 2.0, 2006] as links in the UML object diagram), and the instantiation of attributes (e.g. the attributes in a UML class can have values in the object diagram).

We can distinguish two kinds of instantiation [Atkinson and Kühne, 2001] [Atkinson and Kühne, 2002] [Shavrin, 2007]. Shallow instantiation means that the information is defined on the $n^{th}$ modeling layer and it is used on the immediate instantiation $(n + 1)^{th}$ level. However, in metamodeling it is not always the case. Deep instantiation allows defining an information on the $n^{th}$ modeling level and use it on the $(n + x)^{th}$ ($x > 0$) modeling levels as well. For example, in case of UML Class diagram - Object diagram, the structure of attributes in a class must be defined in the MetaClass diagram, which is the metamodel of all Class diagrams. Class attributes are defined by classes, but the values of the attributes are set in the objects on the Object diagram.

The mechanisms of shallow instantiation have been standardized by OMG [Object Management Group, 2007]. Although there are different approaches to solve the issues of deep instantiations, there does not exist a well-accepted standard in this case. In order to support flexible n-layer metamodeling for DSLs, there is a need for a truly layer-transparent, non UML-based attribute structure that supports deep instantiation and shallow instantiation as well.

Besides solving the instantiation issues, and the ability to create the topological definition of visual languages by using metamodeling, it is also important to be able to define their appearance. A metamodel does not describe the appearance, often called the concrete syntax of its models, thus, there is a need for an additional method to extend metamodel definitions to support this feature.

This chapter presents methods to simplify definition of visual languages created by metamodeling techniques. The chapter focuses on two aspects, namely, on the attributes (attribute structure and instantiation) and on the definition of the concrete syntax. Section 4.1 introduces
the background of the research and main features of existing approaches. Section 4.2 presents an attribute structure, which is truly layer-transparent and supports deep instantiation. The section elaborates the formalism of the attribute structure and the instantiation transformation as well. Section 4.3 presents an efficient approach to define the concrete syntax of visual languages. Finally, a summary is given on the discussed results.

4.1 Backgrounds and Related Work

4.1.1 Attributes of Model Elements

4.1.1.1 Existing Approaches

The primary reference for metamodel-based tools and representations is OMG’s UML Infrastructure and MOF Specification [MOF Infrastructure, 2003]. In the UML Infrastructure, there are four modeling layers (Fig. 4.1).

![Fig. 4.1. The modeling layers in MOF](image)

The meta-metamodeling layer forms the foundation of the metamodeling hierarchy, it is often referred to as M3. According to the standard, the primary responsibility of a metamodel layer is to define a language for specifying models. The metamodel layer is often referred to as M2, models of the M2 layers are instantiations of models on the M3 layer. The modeling level is called M1. A user model is an instance of the UML metamodel. The bottom of the metamodel hierarchy is the M0 layer, which contains the run-time instances of model elements defined in the model. The main limitations of the approach are (i) the fixed number of modeling levels and (ii) the complete lack of layer-transparency. These limitations strongly restrict application possibilities of the approach in n-layer metamodeling.

There are other metamodel-based approaches such as Meta-Modeling Language (MML, [Alvarez et al., 2001]) that aims to define UML models. MML is a static object-oriented modeling language whose focus is the declarative definition of languages. MML aims to create a precise definition of UML, and to enable UML to evolve into a family of languages. However, MML is also based on the four layers of MOF, thus it does not support n-layer metamodeling.

Generic Modeling Environment (GME) [GME Users Manual, 2000] is a popular metamodeling tool. GME refers to the MOF User Object Layer as model level, and the layer above the
Object Layer is called metamodel level, thus, the limitations of UML Infrastructure apply on GME as well.

Similarly to GME, Domain Modeling Environment (DOME) [DOME, 2007] is a meta-CASE system which allows building custom models via specifying their metamodel. The metamodel specification used in DOME is based on different concepts than the MOF specification. However, DOME supports two modeling layers only.

The VIATRA approach [Varró and Pataricza, 2003] introduces the Visual and Precise Metamodelling (VPM), which provides a common, metamodel-based basis for engineering models and graph-like mathematical objects (e.g. abstract state machines, Petri nets). It handles attributes as one of the fundamental modeling types [VIATRA 2, 2005]. Instantiation relationship is represented by special relations such as $instanceOf$, or $typeOf$ instead of an instantiation transformation.

4.1.1.2 Deep Instantiation

In order to support deep instantiation, Atkinson and Kühne proposed to use a new notion [Atkinson and Kühne, 2001] [Atkinson and Kühne, 2002]. The notion allows extending attributes by a potency notion, a number that defines allowed instantiation levels. For example, MetaClass defines the Class model item with potency 2 showing that the model item can be instantiated (on the Class diagrams) and re-instantiated (on the Object diagrams). The notation can be used for model items and attributes as well. Aside from potency, Atkinson and Kühne introduced a dual field notion, where fields describe an attribute and its potency value. Although the approach is mainly based on MOF, it is a straightforward solution to support deep instantiation. The original method does not use a formal specification though. Furthermore, the approach is not fully layer-transparent to the effect that we have to specify the potency value when creating a model item/attribute in the root metamodel instead of specifying it when using the given item/attribute in the concrete metamodel. If the potency value of an attribute changes, all of the modeling layers containing the attribute have to be changed.

Truly layer-transparent deep instantiations have been discussed in [Levendovszky, 2006], where the topological issues have been stressed and solved. However, there is a need to create a similar flexible solution to support n-layer attributes based on shallow and deep instantiations.

4.1.1.3 Representing the Attributes

There are two main approaches in representing the attributes either as attribute nodes (the NodeAttribute approach), or as labels of the model items (the LabelAttribute approach). The second solution requires to handle two different types of nodes in the model: modeling nodes and attribute nodes. In most of the cases (in most of the modeled domains), it is essential to distinguish these two node types. This distinction must be handled by the modeling framework. The other solution, namely storing the attributes in labels does not require such a distinction. The approaches have different advantages and disadvantages:

- In case of the NodeAttribute approach, we can use negative and positive application conditions [Ehrig et al., 2005a] on the attributes, because they are handled as nodes. This
is not true for the LabelAttribute approach, where we require another constraint solving method (e.g. an OCL compiler) in order to be able to handle constraints on attributes.

- The NodeAttribute approach is useful in visualizing attribute structures if model items have only a few attributes. By increasing the number of attributes, models of the NodeAttribute approach become too complex to understand. This limitation does not apply on the LabelAttribute approach, where the two different datastructures (model and attribute) are separated.

- The NodeAttribute approach have difficulties in representing attributes on modeling edges. The issue can be solved by using association nodes (similarly to association classes in UML Class diagrams), but this means that theorems and propositions created for Labeled Directed Graphs cannot be used directly. The issues does not occur in case of the LabelAttribute approach.

- In practice, the NodeAttribute approach is more widely supported. There exists formalisms that describe the approach (for fixed modeling layers and non layer-transparent instantiation). The LabelAttribute approach is harder to formalize if the attributes can be hierarchically nested.

In our approach, we use a hybrid solution, which represents the attributes as labels, but builds a virtual attribute tree from them. Virtual means in this case that the attribute tree is not visualized on modeling diagrams, but it is handled as a hierarchical structure in the memory.

4.1.2 Concrete Syntax

4.1.2.1 Classifying Concrete Syntax Definition Approaches

Without metamodeling, modeling frameworks have to implement a hard-wired set of modeling types with hard-wired visualization. Although metamodeling techniques usually do not support managing the concrete syntax of visual languages directly, there are several techniques to extend metamodeling in order to overcome this shortcoming. These techniques use metamodels as abstract input information and extend this information somehow in order to define the visualization. Although there does not exist a standard for applying this extension, different approaches can be categorized by the method they store the concrete syntax definition and how they apply the extension of the metamodel.

The simplest solution uses the same visualization scheme for all models. This domain-independent notation is a simplified visualization of the abstract representation of the models, referred to as the abstract syntax of a language. The approach does not really extend metamodel definitions, but maps simple graphical objects (e.g. rectangles) to the items of the abstract syntax. The main advantage of the solution is that it does not require additional work when the list of domains increases. If there is a need for an easy to understand representation of the models, but the customization of notation is not important, then this solution is perfect. This is usually the case when creating the metamodel of a domain. Fig. 4.2 shows an example for this, where a class diagram-like representation is used in visualizing a metamodel. It is important to
distinguish the relations by their type (association, or inheritance), but the custom visualization of the nodes is not necessary.

Another way to define the concrete syntax is to extend metamodels with properties focusing on the presentation. This solution means that input metamodels contain properties both for the presentation and the data. Visualization properties are created when the metamodel is constructed. Since presentation logic is encoded in the metamodel items (as properties of the metamodel item), model items with a common metamodel can be treated uniformly. This solution has also its limitations: if the method is not combined with manual coding, then either the customization facilities are strongly limited, or the core framework is extremely complex. Another weakness of the solution is that different models with the same metamodel cannot be visualized differently. The third drawback of the solution is that it does not clearly separate the structural (topological) data and the presentation, which can cause problems of tangled code for the editing process.

Note that there is an alternative of this solution to improve the level of customization. The alternative allows defining presentation-focused properties not only on the metamodel level, but on the model level as well. This approach needs an extension of the original metamodel in order to support defining model level properties. The main drawback of this alternative is that model items have to be customized one by one.

The third solution is to define the concrete syntax not as part of the (meta)model, but as program code applicable in the modeling framework. The approach creates a mapping between the metamodel items and the classes of the code as well. On the one hand, the solution requires manual coding, but on the other hand it is much more flexible, than the first two solutions. Concrete syntax - in this case - can be customized simply, but customization needs a great amount of additional work. Another drawback of the method is that while the previously mentioned solutions can be easily standardized, it is hard to manage in this case. The typical realization of the approach is the plugin-based architecture, where the presentation is coded in plugins assigned to a metamodel and each type defined in the metamodel, has one or more classes defining the visualization.

The fourth solution models concrete syntax definitions as a common Domain-Specific Language (referred to as Presentation DSL), and these definitions are attached to the metamodel. The original metamodel describes the structural (topological) definition of the target domain, while Presentation DSL (which is defined by a metamodel as well) describes the presentation.
Therefore, the different aspects of the domain (data and visualization) are clearly separated. The approach does not need manual coding, it can be standardized, and it allows multiple concrete syntax definitions for a single DSML. Additionally, the solution allows handling the concrete syntax in the same way as the standalone (meta)models, which simplifies editing, improves uniformity and flexibility. Another advantage of the solution is that it allows multiple concrete syntax definitions for a single DSML. Moreover, if the domain is extended by a new type, or if the visualization must be modified, then the concrete syntax can be easily updated for existing and new models as well. In order to improve efficiency, a technique is required to bind the concrete syntax to the structural definitions automatically (e.g., the presentation logic can be transformed to source code that is used to edit the models). The main drawback of the solution is that the presentation functionality is limited by Presentation DSL. This limitation also means that the solution is not applicable in some of the specific domains, where the visualization contains complex interactions between the elements. For example, if the domain is a user interface editor for mobile phones, then the appearance of a button is affected by the current phone, page and maybe by the region settings as well. Such complex interaction is hard to be modeled by the approach (more details in Section 8.3).

In the general case, the target domain does not require support for unique features, but the domain-specific visualization is required. In these cases the Presentation DSL is the most efficient solution due to its advantages mentioned earlier. Moreover, the approach is user-friendly and fast in comparison with full manual coding. Real life examples have shown that creating visualization for a domain like UML Activity Diagram takes approximately two weeks with manual coding, but only one day when using Presentation DSL. This ratio would be ever better when considering the cost of maintenance as well.

### 4.1.2.2 Existing Tools

The Generic Modeling Environment (GME) [Lédeczi et al., 2001] is a highly configurable metamodeling tool supporting two layers: a metamodel, and a modeling layer. The concrete syntax definitions can be coded either manually, or set by properties both on the metamodel and on the model level. GME supports a special type of property definitions: the registry entries. These entries are assigned to model elements and they can also customize the appearance.

Meta-CASE editors (e.g., MetaEdit+ [Meta-case official homepage, 2006]) are environments capable of generating CASE tools. They allow creating the tool definitions in a high-level graphical environment, but they supply a manually coded user interface. These environments store concrete syntax definitions in the metamodel properties.

Another framework is the Diagram Editor Generator (DiaGen) [Minas, 2002], which is an efficient solution to create visual editors for DSLs. DiaGen is not based on metamodeling techniques; it uses its own specification language for defining the structure of diagrams. DiaGen supports editing the concrete syntax in a graphical context, but in a tree control-based form only, where there is no support to define the shape of the elements graphically. Concrete syntax in DiaGen is based on properties. DiaGen can generate an editor based on the specification using hypergraph grammars and transformations, but the generated modeling tools are not meant to
be integrated into existing modeling frameworks. This restriction and the specialities of the specification language limit the usability and maintenance capabilities of DiaGen.

AToM3 (A Tool for Multi-formalism and Meta-Modelling) [de Lara and Vangheluwe, 2002]) is a flexible modeling tool. It employs an appearance editor to define the shape of the model elements graphically; it uses model level properties to store the concrete syntax (model definitions are extended by visualization-based attributes). AToM3 can generate plug-ins that use the defined syntax, but the code generation is not based on a Presentation DSL and the generated code implements a stand-alone executable (not a plugin). The views of the models are generated with triple graph grammars.

Eclipse [Budinsky et al., 2003] is probably the most popular, highly flexible, open source modeling platform that supports metamodeling. The Eclipse Modeling Framework (EMF) can generate source code from models defined by the class diagram definition of UML, but it does not contain concrete syntax definitions (EMF uses abstract syntax). Moreover, EMF can generate a tree-based editor in which models can be edited by defining symbols and values for presentation-based properties. Only tree-based editing is supported, which is not always enough. The Graphical Editing Framework (GEF) is also a part of the Eclipse project. GEF provides methods for creating visual editors. EMF does not support code generation for GEF, therefore GEF plug-ins require manual coding to support the concrete syntax.

GenGed [GenGed - Homepage, 2005] is a tool to generate visualization code with graph transformation. GenGed has been replaced by the project Transformation-Based Generation of Modeling Environments (TIGER) [Ehrig et al., 2005b] that uses precise visual language (VL) definitions and offers a graphical environment based on GEF. TIGER can generate source code from the visual language definitions, this source code implements a plug-in based on GEF. Main advantages of TIGER are that it supports mapping VL specifications to metamodels and VL specifications can be created graphically. Java is the only language supported in plug-in generation. At the moment TIGER can generate editors only for Activity Diagrams and Petri nets.

The Graphical Modeling Framework (GMF) is also an Eclipse project. The goal of GMF is to form a generative bridge between EMF and GEF, whereby a diagram definition is linked to a domain-specific model as an input to the generation of a visual editor. GMF uses the metamodel definition of EMF and a Presentation DSL to define the concrete syntax. The result (the mapped concrete, and structural definitions) are processed further to produce source code. The mapping between the domain model and the model items of the concrete syntax is also supported in GMF. The generated source code relies on the features of GEF and EMF. Although the concept of GMF is straightforward, it has some weaknesses: (i) GMF does not support checking whether a specific concrete syntax has been created for the target domain. (ii) The concrete syntax definition can be defined in a semi-graphical way only. Although the graphical preview of the appearance definition is always available, considerably large part of the definition must be defined textually. (iii) The generation is not based on model transformation. Consequently, the compilation steps are coded manually, thus, changing the transformation needs changing the source code and rebuilding the compiler. In case of model transformation such modifications can be accomplished at run-time. (iv) Because of EMF, GMF is restricted to Java only.
As it has been shown, although the appearance definition is of great importance in visual model creation, *Presentation DSL* is supported only in a few modeling tools and even in these tools, there are issues to solve, real graphical construction of appearance definition is not supported. There is a real need for a metamodeling approach that supports defining the concrete syntax based on *Presentation DSL*.

### 4.2 N-Layer Attribute Structure

The following subsections introduce a completely new attribute structure, which can be used in creating visual languages. The attribute structure and the transformation used in instantiation relationship are based on formal syntax. Firstly, a simple algebra and transformation are presented that support shallow instantiation only, then they are extended to handle deep instantiated attributes as well. In order to distinguish the approaches we refer to them as Simplified N-Layer Attribute Algebra and Enhanced N-Layer Attribute Algebra.

#### 4.2.1 Attribute Structure

The basic attribute structure of the approach consists of three elements: *InstanceName*, *Attribute*, and *ComplexType*. These elements are available at every model layer. The *InstanceName* tag contains the name of the type defined by the item. *Attribute* can define an attribute for the model level, while *ComplexType* is used to express complex data types for attributes.

*ComplexTypes* are structure definitions only, they need always an *Attribute* to appear as an attribute on the model level. *Attributes* are defined by tags describing the details of the attribute. The tag *Name* contains the name of the attribute; *TypeExpression* describes the type of the attribute. The *TypeExpression* is either a primitive type, or the name of a *ComplexType*. The tag *Multiplicity* contains the minimum and maximum number of the attribute instances, for example 0..*. The *ComplexType* definitions contain the structure of non-primitive attribute types. *ComplexTypes* are also defined by tags. The *Name* tag contains the name of the type, while the structure of the type is handled as a composition of primitive types. Therefore, *ComplexType* can contain several *Attributes* defining the structure. *ComplexTypes* can be nested.

**Example 4.1.** In the MetaClass diagram, there is a model item with an *InstanceName* ”Class”. This model item defines several attributes, such as *ClassAttribute* and *ClassOperation*, and complex types to describe the attributes, for example *ClassAttributeComplexType*. *ClassOperationComplexType* consists of *OperationName*, *ReturnType*, list of *Arguments*, and *Visibility*. *OperationName* is a string value, while *Argument* uses another complex type defining the name, and type of the argument.

The defined attribute structure of the metamodel is transformed to a validation schema for the model level. Model level attributes cannot modify this schema in general, but they can extend it by defining new *Attributes* or *ComplexTypes*. 

4.2.2 The Simplified N-Layer Attribute Algebra

Simplified N-Layer Attribute Algebra is the formal description of the discussed attribute structure. The formalization is based on Abstract State Machines.

The algebra represents the attributes of a model item in a tree structure, where the root of the tree is the model item itself. All model items and attributes have a unique identifier (ID) in order to make the identification and reference handling simpler.

The current attribute configuration of a model item is represented using shared functions. The values of the functions are changed either by the algebra itself, or by the environment of the algebra (for example by the user). A state of the algebra represents an attribute configuration, namely, a concrete set of parameter set-value pairs for the functions. For example, there is a state \( S_1 \) in which we have a model item \( m_1 \) with no attributes defined. If a new attribute is defined or an old attribute is modified/deleted, then the value of the functions and the current state of the algebra change. A possible next state of \( S_1 \) is \( S_2 \), which contains the same model item, but in this case \( m_1 \) has an attribute \( a_1 \).

The function \( \text{Meta}(ID) \) returns the meta item/attribute of the model item/attribute obtained as parameter. The value of the function changes when a new model item/attribute is created. The function \( \text{IsAttribute}(ID) \) is used to differentiate between attribute and model item IDs; the function returns true, if the parameter is the ID of an attribute. The function \( \text{Name}(ID) \) results in the name of the model item, or attribute retrieved as parameter. The function \( \text{Root}(ID) \) obtains the root of the attribute tree, namely the model item. The function \( \text{Attribute}(ID, N, I) \) is used to obtain the value of an attribute. The first parameter selects the model item, or the attribute \( p \) which is the parent of the attribute that we try to find. Then, the function collects the children nodes of \( p \) that have the name defined in the second parameter. The third parameter selects the \( I^{th} \) element from this list. If \( p \) cannot be found, or it does not have the children requested, the function returns \text{undef}.

**Definition 4.2.** The superuniverse \( |A| \) of a state \( A \) of the Simplified N-Layer Attribute Algebra is the union of four universes:

- \( U_{\text{Bool}} \) containing logical values \{true/false\}
- \( U_{\text{Number}} \) containing rational numbers \( \{\mathbb{Q}\} \)
- \( U_{\text{String}} \) containing character sequences of finite length
- \( U_{ID} \) containing all the possible IDs.

The precise definition of the aforementioned functions is based on labels attached to the model items and edges representing the model. This model is a Labeled Directed Graph (LDG, see Section 3.3 for the definition). For the sake of simplicity, we handle labeling functions \( lv^G \) and \( le^G \) by a dual field notation representing \( Name / Value \) pairs. We refer to a label with the name \( N \) of the model item \( X \) as \( X_N \).

We define the following special labels:

- For each node (attribute and model item) and edge \( X \) we define
– a globally unique label $X_{ID}$ that takes its value from $U_{ID}$.
– a label $X_{ModelItem}$ that takes its value from $U_{Bool}$.
– a label $X_{TypeOf}$ that takes its value from $U_{ID}$.
– a special, complex label $X_{ContainedAttributes}$ that contains a possibly empty, indexed list of $U_{IDS}$. The index is an ordinal number representing the order of attributes.
– a label $X_{Value}$. The label returns $undef$ if the node is not an attribute with simple type, otherwise it contains the value taken from $U_{Bool}$, $U_{Number}$, $U_{String}$, or $U_{ID}$.

Our approach uses a hybrid solution in storing attributes. This hybrid solution has to specialities: (i) virtual nodes and (ii) the $X_{ContainedAttributes}$ label. In order to simplify the formalization, we store attributes in virtual attribute nodes. Attributes are handled as virtual nodes that means that in practice, they do not appear as real (modeling) nodes in diagrams. However, by introducing virtual nodes, we can create functions that can process usual model items and attributes as well, thus, we do not need to duplicate the functions.

The second speciality is the label $X_{ContainedAttributes}$ that is a list of attribute IDs. Recall that IDs are unique, thus, this list is similar as if we would have pointers, or links to the contained attributes. Obviously it would be simpler to connect virtual attribute nodes directly to real model items, but in metamodeling, edges can also have attributes. However, the definition of LDG does not allow connecting edges to edges (the result of src/target function is always a node). We have solved the issue by using this pointer list as label. Note that attributes can also have other attributes as children, thus, $X_{ContainedAttributes}$ is used in case of virtual nodes as well.

Moreover, in order to simplify handling the $X_{ContainedAttributes}$ label, we use the following algorithm to handle the indexed list:

\begin{algorithm}
\caption{The \texttt{ATTRIBUTEHANDLER} function}
\begin{algorithmic}[1]
\State \texttt{ATTRIBUTEHANDLER (Parent,Name,Idx)}
\State \texttt{Counter = 0;}
\ForAll {ChildID in Parent$X_{ContainedAttributes}$}
\State \texttt{CurrentItem: CurrentItem\_ID = ChildID}
\If {Name(CurrentItem) = Name} \texttt{Counter = Counter + 1} \EndIf
\If {Counter = Idx} \texttt{return CurrentItem} \EndIf
\EndFor
\State \texttt{return undef}
\end{algorithmic}
\end{algorithm}

Using these labels we can define the monitored functions:
### Definition 4.3

The vocabulary Σ of the Simplified N-Layer Attribute Algebra formalism is assumed to contain the following characteristic functions (arities are denoted by slashes): Meta/1, IsAttribute/1, Name/1, Root/1, Attribute/3.

### 4.2.3 Valid and Invalid Models

The Simplified N-Layer Attribute Algebra distinguishes valid and invalid models, where checking validity is based on formulas describing different properties of the model. When external systems change the attribute tree, using the functions of the algebra, these formulas must hold.

In order to simplify the definition of the formulas, a helper function \( \varphi_{\text{IsUnique}}(ID, Name) \) is introduced.

\[
\varphi_{\text{IsUnique}}(ID, Name) = \begin{cases} 
true, & \text{if } \exists I_1 \land \exists I_2 : \text{Attribute}(ID, Name, I_1) <> \text{undef} \land \\
& \text{Attribute}(ID, Name, I_2) <> \text{undef} \land I_1 <> I_2 \\
false, & \text{otherwise}
\end{cases}
\]

The function \( \varphi_{\text{IsUnique}}(ID, Name) \) shows whether the Name attribute is unique among the child attributes of the model item/parent attribute defined by ID. Using this helper function, the following properties can be defined: (i) each model item has a unique InstanceName attribute, (ii) each attribute with the name 'Attribute' has Name, MinMul, MaxMul and Type children; these children are unique, (iii) two Attribute definitions with the same Name cannot exist, (iv) each attribute with the name 'ComplexType' has a Name child and one or more Attribute children, (v) two complex type definitions with the same Name cannot exist, (vi) Attributes and ComplexTypes cannot use the reserved words of the algebra as Name. Valid models fulfill all the six conditions.
4.2.4 The Transformation

In order to solve the issues of n-layer attribute handling, it is not enough to create an attribute structure: an instantiation transformation is required as well. The instantiation transformation is described by a rule of the algebra. When creating a new model item, this rule is used to initialize the attribute structure of the item according to the attributes defined in the metamodel.

To simplify the definition of the instantiation rule, we define helper functions as follows:

\[
\varphi_{\text{ InstName }} = \begin{cases} 
\text{true}, & \text{if } \forall ID : \text{if}(\neg \text{IsAttribute}(ID)) \to \varphi_{\text{IsUnique}}(ID,'InstanceName') \\
\text{false}, & \text{otherwise}
\end{cases}
\]

\[
\varphi_{\text{ Attr }} = \begin{cases} 
\text{true}, & \text{if } \forall ID : \text{if}(\text{Name}(ID) = 'Attribute') \to (\varphi_{\text{IsUnique}}(ID,'Name') \land \\
\varphi_{\text{IsUnique}}(ID,'MinMul') \land \varphi_{\text{IsUnique}}(ID,'MaxMul') \land \\
\varphi_{\text{IsUnique}}(ID,'Type')) \\
\text{false}, & \text{otherwise}
\end{cases}
\]

\[
\varphi_{\text{ AttrName }} = \begin{cases} 
\text{true}, & \text{if } \forall ID_1 : \text{if}(\text{Name}(ID_1) = 'Attribute') \to (\exists ID_2 : \\
\text{Name}(ID_2) = 'Attribute' \land ID_1 <> ID_2 \land \\
\text{Attribute}(ID_2,'Name',0) = \text{Attribute}(ID_2,'Name',0)) \\
\text{false}, & \text{otherwise}
\end{cases}
\]

\[
\varphi_{\text{ CplxType }} = \begin{cases} 
\text{true}, & \text{if } \forall ID : \text{if}(\text{Name}(ID) = 'ComplexType') \to \\
(\varphi_{\text{IsUnique}}(ID,'Name')) \land \\
(\exists Idx : \text{Attribute}(ID,'Attribute',Idx) <> \text{undef}) \\
\text{false}, & \text{otherwise}
\end{cases}
\]

\[
\varphi_{\text{ TypeName }} = \begin{cases} 
\text{true}, & \text{if } \forall ID_1 : \text{if}(\text{Name}(ID_1) = 'ComplexType') \to (\exists ID_2 : \\
\text{Name}(ID_2) = 'ComplexType' \land ID_1 <> ID_2 \land \\
\text{Attribute}(ID_2,'Name',0) = \text{Attribute}(ID_2,'Name',0)) \\
\text{false}, & \text{otherwise}
\end{cases}
\]

\[
\varphi_{\text{ ReservedNames }} = \begin{cases} 
\text{true}, & \text{if } \forall ID : \text{if}(\text{Name}(ID) = 'Attribute', 'ComplexType') \to \\
(\text{Attribute}(ID,'Name',0) \notin \\
\{ 'InstanceName', 'Attribute', 'ComplexType' \}) \\
\text{false}, & \text{otherwise}
\end{cases}
\]

\[
\varphi_{\text{ ValidModel }} = \varphi_{\text{ InstName }} \land \varphi_{\text{ Attr }} \land \varphi_{\text{ AttrName }} \land \varphi_{\text{ CplxType }} \land \varphi_{\text{ TypeName }} \land \varphi_{\text{ ReservedNames }}
\]

The function \( \text{GetAttributes}(P) \) obtains the children of the selected model item/attribute \( P \), with the name 'Attribute'. The function \( \text{ChildCounter} \) counts how many children the selected parent \( P \) has with the given name \( N \). The function \( \text{Add} \) creates a new child attribute for \( P \) in the attribute tree, where the name of the new attribute is set to \( N \). The function \( \text{ComplexAttribute} \)
is used to check whether an attribute is of simple type. The function \( \text{GetComplexType} \) retrieves the first \( \text{ComplexType} \), which is defined in the model item \( \text{Root}(A) \) and have the name \( N \).

**Algorithm 4.2 The TransformAttributes rule**

1. rule \( \text{TransformAttributes}(ID, ID_{\text{Meta}}) \)
2. if \( \neg \text{IsAttribute}(ID) \) then
3. \( \text{Name}(ID) = \text{ATTRIBUTE}(ID_{\text{Meta}}, 'InstanceName', 0) \)
4. \( \text{Add}(ID, 'InstanceName') \)
5. for all \( A \) in \( \text{GetAttributes}(ID_{\text{Meta}}) \) do
6. \( N = \text{ATTRIBUTE}(A, 'Name', 0) \)
7. \( \text{for } i = 0 \text{ TO } \text{ATTRIBUTE}(A, 'MinMul', 0) \) do
8. \( \text{Child} = \text{Add}(ID, N) \)
9. \( \text{META}(\text{Child}) = A \)
10. if \( \text{ComplexAttribute}(A) \) then
11. \( \text{TransformAttributes}(\text{Child}, \text{GetComplexType}(A, \text{ATTRIBUTE}(A, 'Type', 0))) \)

Using these functions, the instantiation transformation can be defined. Recall that this transformation supports shallow instantiated attributes only. The instantiation algorithm is shown in Alg. 6.1. The first parameter of the rule \( \text{TransformAttributes} \) is the local root, this is the model item, or the attribute on the instantiation level, whose children we process. The second parameter is the metaitem, or attribute associated with the local root.

If the local root is a model item, then the name of the model item is set to \( 'InstanceName' \) attribute of the metamodel item. Moreover, an empty \( 'InstanceName' \) attribute is added to the model item automatically in order to obey the rules of valid models.

In the next step, the list of \( \text{Attributes} \) are traversed and processed one-by-one. The current attribute definition in the metamodel is selected by \( A \). The rule creates attributes \( (\text{Child}) \) according to the minimum multiplicity \( ('MinMul') \) property of \( A \).

Note that the number of descendant levels of an attribute in the attribute tree is not limited by the algebra. The children of an attribute can have children as well.

**Example 4.4.** The attribute \( a_1 \) is of complex type \( c_1 \). \( C_1 \) has two sub-attributes \( a_2 \) and \( a_3 \). The Simplified N-Layer Attribute Algebra does not limit nesting of complex types, the type of \( a_2 \) or \( a_3 \) can be a complex type as well. In the instantiated model, attributes created from \( a_1 \) have two children attributes as defined in \( (a_2, a_3) \) and they can have grandchildren or other descendants as well.

Therefore, if the type defined in \( A \) is a complex type, then the rule processes the definition of the type recursively.

The presented transformation rule can create a basic, initial and valid attribute structure for model items. Note that this attribute structure can be modified later, both by external systems (representing for example the user), or by other, internal algorithms as long as the formula \( \varphi_{\text{ValidModel}} \) remains true.

UML class models have an instantiation definition standard, which is applicable only to class diagrams. In case of UML, we cannot use the same instantiation in class diagrams and for
example in statechart diagrams. In contrast, the rule TransformAttributes can be used in a transparent way between modeling layers. It does not need any special information or property of the metamodel in order to be applied. However, the rule is intended to process only valid metamodels, where validity is defined by the $\phi_{ValidModel}$ formula.

4.2.5 Supporting Deep Instantiation

4.2.5.1 Extending the Attribute Algebra

The Simplified N-Layer Attribute Algebra does not support deep instantiation. In order to handle deep instantiated attributes, we use a similar solution to [Atkinson and Kühne, 2001]. However, instead of global, numeric potency values, our approach is based on a triple: the attribute is extended by a Boolean field that allows or deny re-instantiation, and a string field that defines the name of the re-instantiated attribute. Instead of single attribute, we always use a triple following the described structure. Note that attributes of the Simplified N-Layer Attribute Algebra cannot be transformed always automatically to the triple form, because deep instantiation is driven by the user or other external mechanisms. The new (extended) approach is referred to as the Enhanced N-Layer Attribute Algebra.

Example 4.5. An attribute Name of Simplified N-Layer Attribute Algebra is transformed to a triple of Enhanced N-Layer Attribute Algebra. The result triple is \{Name, False, "\}, which means that the attribute does not appear on the instantiation level.

Another attribute Timer is transformed as well, the triple \{Timer, True, 'SynchronousTimer'\} defines in this case that the attribute is available in the instances of the model item under the name 'SynchronousTimer'.

The main advantages of our approach over classical potency values [Atkinson and Kühne, 2001] are that (i) a re-instantiation of an attribute to the $n + 1^{th}$ level is set on the $n^{th}$ level, not at the first modeling level the attribute appears; (ii) we can use different names on the instance level and on the metamodel level. The first advantage means that only the direct metamodel has to be changed in order to re-instantiate an attribute. The second advantage allows us differentiating attributes of a metamodel with the same name. For example, if the meta-metamodel item defines that the metamodel item can have several Timers, then it can be useful to distinguish these Timers on the instance level for example as 'SynchronousTimer', 'AsynchronousTimer', or 'GlobalTimer'.

In order to support the triple structure, the functions and the algebra must be changed. More precisely, for sake of simplicity, we define the functions Meta, IsAttribute, Name, Root, Attribute to retrieve always the first component of the triple. Thus, they implement the same functionality as in Simplified N-Layer Attribute Algebra. However, to handle the second and third component of the triple, we use two new functions: ReInstantiate(ID) and ReInstantiationName(ID). The first function is a shared function, it has one input parameter, the ID of the attribute. The function returns a Boolean value, namely the second component of the triple. The function ReInstantiationName(ID) works similarly, but it returns the third component of the triple. To support the new functions, we define the following labels:
• For each node (attribute and model item) and edge \(X\) we define
  
  – a label \(X_{\text{ReInst}}\) that takes its value from \(U_{\text{Bool}}\).
  
  – a label \(X_{\text{ReInstName}}\) that takes its value from \(U_{\text{String}}\).

The definition of the new monitored functions is the following:

\[
\begin{align*}
\text{ReInstantiate}(ID) & : \begin{cases} 
B, & \text{if } \exists X: X_{ID} = ID \land B = \neg X_{\text{ReInst}} \\
\text{undef}, & \text{otherwise}
\end{cases} \\
\text{ReInstantiationName}(ID) & : \begin{cases} 
\text{Name}, & \text{if } \exists X: X_{ID} = ID \land \text{Name} = \neg X_{\text{ReInstName}} \\
\text{undef}, & \text{otherwise}
\end{cases}
\end{align*}
\]

**Definition 4.6.** The vocabulary \(\Sigma\) of the Enhanced N-Layer Attribute Algebra formalism is assumed to contain the following characteristic functions (arities are denoted by slashes): Meta/1, IsAttribute/1, Name/1, Root/1, Attribute/3, ReInstantiate/1, ReInstantiationName/1.

Note however, that the superuniverse of the algebra does not change:

**Definition 4.7.** The superuniverse \(|\mathfrak{A}|\) of a state \(\mathfrak{A}\) of the Enhanced N-Layer Attribute Algebra is the union of four universes:

- \(U_{\text{Bool}}\) containing logical values \{true/false\}
- \(U_{\text{Number}}\) containing rational numbers \(\mathbb{Q}\)
- \(U_{\text{String}}\) containing character sequences of finite length
- \(U_{ID}\) containing all the possible IDs.

### 4.2.5.2 Extending the Instantiation Transformation

The second and third component of the triple are handled by the instantiation transformation in case of built-in tags (\(\text{InstanceName}\), \(\text{Attribute}\) and \(\text{ComplexType}\)). Therefore, triple definitions for these tags cannot be set by external systems or users. More precisely, they can be set, but the following formula must be remain true:

\[
\varphi_{\text{ReInst}} = \begin{cases} 
\text{true, if } \forall ID: & (\text{Name}(ID) = \prime \text{InstanceName'} \rightarrow (\neg \text{ReInstantiate}(ID))) \lor \\
& (\text{Name}(ID) = \prime \text{Attribute'} \rightarrow (\text{ReInstantiate}(ID)) \land \\
& \text{ReInstantiationName}(ID) = \text{Attribute}(ID, \prime \text{Name'}, 0)) \lor \\
& (\text{Name}(ID) = \prime \text{ComplexType'} \rightarrow \neg \text{ReInstantiate}(ID)) \lor \\
\text{false, otherwise}
\end{cases}
\]

The formula \(\varphi_{\text{ValidModel}}\) is also modified to contain the new condition:

\[
\varphi_{\text{ValidModel}} = \varphi_{\text{InstName}} \land \varphi_{\text{Attr}} \land \varphi_{\text{AttrName}} \land \varphi_{\text{CplxType}} \land \varphi_{\text{Name}} \land \varphi_{\text{ReservedNames}} \land \varphi_{\text{ReInst}}
\]

Using the presented extension in the algebra, we can create the instantiation transformation that supports deep instantiation. We use a helper function \(\text{GetChildren}\) to simplify the extended transformation rule.
The extended instantiation algorithm is shown in Alg. 4.3. The algorithm checks, whether the parameter (the local root) is an attribute, or a model item. If it is a model item, then the algorithm creates the InstanceName attribute in order to obey the formulas of valid attribute configuration. If the local root is an attribute, then the algorithm checks whether it is an Attribute, or an already instantiated attribute. Attributes are processed recursively similarly as in Alg. 6.1, the only difference is the initialization of ReInstantiate and ReInstantiationName functions. If the local root is an already instantiated attribute and the attribute is marked as reinstantiatable, then a new attribute is created with the name defined by ReInstantiationName. Reinstantiaatable attributes must also be processed recursively, because reinstantiation does not flatten the attribute hierarchy.

Algorithm 4.3 The TransformAttributes2 rule

1: rule TransformAttributes2(ID, IDMeta)
2: if (¬ ISATTRIBUTE(ID)) then
3: NAME(ID) = ATTRIBUTE(IDMeta,'InstanceName',0)
4: Child = ADD(ID,'InstanceName')
5: ReInstantiate(Child) = false
6: for all A in GetChildren(IDMeta) do
7: if (NAME(A) = 'Attribute') then
8: N = ATTRIBUTE(A,'Name',0)
9: for i = 0 TO ATTRIBUTE(A,'MinMul',0) do
10: Child = ADD(ID,N)
11: Meta(Child) = A
12: ReInstantiate(Child) = false
13: ReInstantiationName(Child) = ”
14: if COMPLEXATTRIBUTE(A) then
15: TransformAttributes2(Child,
16: GetComplexType(A, ATTRIBUTE(A,'Type',0)))
17: else
18: if (ReInstantiate(A)) then
19: Child = ADD(ID, ReInstantiationName(A))
20: ReInstantiate(Child) = false
21: ReInstantiationName(Child) = ”
22: for all A2 in GetChildren(A) do
23: TransformAttributes2(Child,A2)

4.2.6 Properties of the Enhanced N-Layer Attribute Algebra

Definition 4.8. The depth of a model item is the maximum number of nesting levels of complex types in the model item.
Proposition 4.9. The instantiation transformation for a model $M$ requires

$$
\mathcal{O} \left( \sum_{i=1}^{m} (n_i + r_i \cdot c_i) \cdot (a_i \cdot c_i)^{\text{depth}_i} \right)
$$  \hfill (4.1)

model queries in the worst case, where $m$ is the number of model items in $M$, $n_i$ is the number of attributes on the uppermost level of the $i^{th}$ model item, $r_i$ is the number of re-instantiated attributes, while $c_i$ is the maximum number of components (children) in an attribute of the model item, $\text{depth}_i$ is the depth of the $i^{th}$ metamodel item, and $a_i$ is the highest minimum attribute cardinality used in Attributes of the $i^{th}$ model item.

Proof. Appendix A.

Note that the number of attributes in the model item is an upper bound for $n_i$, $c_i$ and $r_i$. Since Proposition 4.9 gives an upper bound for the number of steps, thus, the instantiation requires finite number of steps (finite number of model queries) if the depth of the model items are finite and the model items contain finite number of attributes.

Definition 4.10. Semi-free instantiation of a valid model $M$ means the application of $\text{TransformAttribute}_2$ on $M$ and modifying the result model $M_I$ by not breaking the validity rules, namely, the $\varphi_{\text{ValidModels}}$ formula.

The effects of semi-free instantiation can be described by rule of the algebra that composes the effects of the instantiation and the modifications done after the instantiation. The application of this rule is referred to as a function $f_{\text{SFInst}}$.

Proposition 4.11. Given a valid root metamodel $M_0$, the n-ary composition of $f_{\text{SFInst}}(M_0)$, namely $f^n_{\text{SFInst}}(M_0)$ exists for all finite $n$ and it produces a valid model $M_N$.

Proof. Appendix A.

By combining the statements of Proposition 4.9 and Proposition 4.11 we can see that the presented instantiation method is always applicable in anarbitrary depth and the application requires finite number of model queries, if the conditions of the propositions are satisfied. Since these conditions are naturally satisfied in real life examples, the approach realizes an applicable solution to handle attributes of an n-layer metamodeling system.

4.3 Defining the Concrete Syntax

Our approach on defining the appearance of visual languages is based on a Presentation DSL, namely on the VMTS Presentation DSL (VPD). Fig. 4.3 shows an overview of the relationships between the structural metamodel definition and the concrete syntax of the target language.

The metamodel of our VMTS Presentation DSL is a special metamodel created to describe concrete syntax definitions. In other words, the metamodel defines a palette to create the concrete syntax. The metamodel is referred to as VPD Metamodel. By instantiating the metamodel, VPD models can be created. A VPD model contains customized visualization for model items...
Chapter 4. Defining Visual Languages

4.3. Defining Visual Languages

4.3.1 A Metamodel for Visualization

The VPD metamodel consists of five nodes as shown in Fig. 4.4. Appearance Definition can describe the graphical notation of the model elements. Weaving Contexts are used to define behavioral attributes and to store mapping information between the concrete syntax and the structure definition.

---

Fig. 4.3. Concrete Syntax Definition - Overview

Fig. 4.4. VMTS Presentation DSL - Metamodel
The name *Weaving Context* describes that these elements weave two different aspects of the model, the data and the visual definition, namely the structural definition and the concrete syntax. For example, in case of control flow diagrams *Appearance Definitions* define the graphical notation, such as rectangle for statement, or diamond for conditions. *Weaving Contexts* have a reference to the appropriate metamodel item, thus, *StatementContext* has a reference to *MetaStatement* item in the control flow metamodel. *Weaving Contexts* also contain several behavioral attributes, such as the minimum size of the model element. A relation between *Weaving Context* and *Appearance Definitions* can be constructed using *Attribute Reference* relations. In the metamodel, the multiplicity of this relation is many-to-many, which means that the appearance definitions are reusable, and the weaving contexts can have several appearances. This reusability is necessary because modeling languages have a tendency to use the same notation in different languages. For example, *StartState* in UML 2.0 statechart diagrams and *InitialState* in UML 1.4 activity diagrams are denoted the same way. Similar separation between the behavioral attributes and the mapping information could be created, but we have found that customized behavioral attributes are harder to reuse. For example, *Input* pins should be aligned to one side of its container in UML activity diagrams. This property is handled by behavioral attributes (positioning constraints). The constraints describe the alignment rules useful only for this type of elements.

Different fundamental types can have different behavioral and visualization properties, thus, they are distinguished. In our metamodeling approach, there are three fundamental types: *Nodes, Edges* (relations between nodes) and *AssociationNodes* (e.g. *AssociationClass* in UML class diagrams). The VPD metamodel mirrors these fundamental types by customizing *General Weaving Context*. For example, the model item *NodeWeavingContext* can express the mapping information for a *Node*.

Visualization information described in *Appearance Definitions* is based on *regions*. A region is responsible for visualizing a part of the model item, or the whole model item. Regions are graphical units that are independent from each other, thus, they can be edited separately, and the model representation can be composed of the regions when displaying the model item. Region definitions consist of simple graphical objects called *primitives* and *extended primitives*. Primitives are lines, Bézier splines, triangles, rectangles, rounded rectangles, diamonds, ellipses, crossed circles, and static text labels. Extended primitives are for example dynamic texts that support displaying the properties of the model items. A region can contain several primitives, for example an *Actor* in a *Use-Case* model is defined in a single region although it consists of several primitives (head, body, arms, legs). Primitives support location constraints as well, which are handled by alignment information. This alignment information shows how the appropriate primitive is positioned in the region and how it behaves if the region is resized similarly to position constraints on controls of a form.

### 4.3.2 A Model Transformation for Processing the Concrete Syntax

Model transformations can be modeled in a visual way, they can be changed easily and they can use the efficient graph transformation techniques, along with high level transformation constraints. That is why our approach uses model transformation to process the concrete syntax.
Model transformation obtains the input model and metamodel, the output metamodel, and the control flow model which defines the transformation steps, while it produces the result of the transformation in the output model. The input model is an instance of the input metamodel, and the output model is an instance of the output metamodel. Fig. 4.5 shows an overview of model transformation in case of VMTS Presentation DSL.

In our approach, Concrete Syntax Models, namely, the concrete syntax definitions, are transformed to CodeDOM models [Thai and Lam, 2003]. CodeDOM supports describing source code in an abstract form, thus, it is a practical basis to transform the appearance definition to. Recall that in our model transformation approach, the control flow for the transformation can be constructed using the Visual Control Flow Language (VCFL) [Lengyel et al., 2006i]. The transformation nodes in the control flow specify the operational behavior of model processing. These nodes are graph rewriting rules. Besides the rewriting rules, VCFL also supports decisions, fork, and join items and parameter passing between the subsequent transformation rules. The control flow of the model transformation is shown in Fig. 4.6.

The first rewriting rule (CreateNamespace) is an initialization step for the further operations, it creates an environment for the generated output, for the source code in this case. The concrete
syntax definition is processed in the steps *GetUnprocessedNode* and *MatchAppearances*. These steps are connected in a loop using a decision item. In VCFL, the decision steps can use OCL constraints, or simply the result of the previous rewriting steps to decide on which branch they continue the execution. In this case the loop exits only if there are no unprocessed node (weaving context) left in the concrete syntax definition, which means that the step *GetUnprocessedNode* was unsuccessful. The step *GetUnprocessedNode* is simple: both LHS and RHS contains a *General WeavingContext* node. Matching information is accomplished using an OCL constraint and a *Virtual Attribute*. Virtual attribute is a special, temporary attribute added to the matched elements during model transformation [Mezei et al., 2006i]. Using virtual attributes, the original model items can be decorated without changing their meta definitions. Virtual attributes are removed at the end of the transformation. In this case the rule *GetUnprocessedNode* is based on the virtual attribute *IsProcessed*. The weaving context in LHS is extended by an OCL constraint that ensures that the matched node has not been matched before. The rule also contains a modify type internal causality. Internal causality is a relationship between LHS and RHS nodes, they define the attribute computations. This causality adds the *IsProcessed* attribute to the matched node. The rule *MatchAppearances* is more complex (Fig. 4.7).

![Image](image.png)

**Fig. 4.7.** The rule *MatchAppearances*

The rule matches the weaving context along with the associated appearance definitions. The *Context* element of LHS is passed to the rule from the *GetUnprocessedNode* rule using the parameter passing mechanism. The matching algorithm considers these assignments compulsory. A single weaving context can have multiple appearances as mentioned before, thus, the relation has a multiplicity of 1..*. From the matched context and appearance, model items are generated into the output model by RHS. The rule consists of create type internal causalities only, which means that no model element is defined or deleted by the rule.

The main loop of the transformation exits if the input model is fully processed. The step *ClearHelperInformation* deletes the *IsProcessed* attribute from the weaving contexts. The rule is defined as a *MultipleMatch* rule, which means that it is applied for each weaving context in
the host model. EndNode supports a special type of action, After Action is executed at the end of the transformation. This special action is used to process the output model and create source code in our approach.

4.3.3 Properties of the Transformation

Using a model transformation to process the concrete syntax definition is a straightforward solution, because the changes in the modeling methods or in the modeling structure can be easily adopted. ‘Easily’ means that only the transformation control flow and the rewriting rules need to be modified. In contrast, classic model compilers would fail, for example, if a new fundamental type is required. This flexibility has also some drawbacks: if the transformation changes, then its correctness must be proven again. Using constraints in transformation rules can help in creating a validated model, but there are transformation-level properties, such as the question of termination, which require further examination. The aim of our analysis is to prove that the transformation terminates for every valid finite input model. We use the definitions and theorems presented in [Levendovszky et al., 2006] to make the proof method simpler. These theorems are proven for injective rules only, but this is not a problem, since the VPD transformation uses injective matches.

Using the existing theoretical results (Section 3.3.5), we can prove the following proposition:

Proposition 4.12. The transformation VPD depicted in Figure 4.6 always terminates.

Proof. Appendix A.

4.4 Chapter Summary

Visual languages are frequently used in model-based development and software engineering projects. The ability to create and customize these languages in a flexible way is one of the key features in creating a modeling environment. Although metamodeling is an efficient way to define languages, the current metamodeling techniques have several shortcomings as well.

The chapter has proposed two constructs to simplify the construction of visual languages: an attribute algebra to handle n-layer attributes, and a special visual language-based approach to construct appearance definitions.

The chapter has elaborated the shortcomings of existing attribute modeling approaches. Then a new formal attribute structure (Simplified N-layer Attribute Algebra) has been presented. The attribute structure is not restricted to UML-based attributes, it offers a much more general and much more flexible method to define the attributes. The formal description of the structure ensures preciseness of the approach. Formulas have been given to distinguish valid and invalid models. Moreover, an instantiation method has been presented that supports transforming the attribute structure of metamodel items to their instance regardless on which metamodeling layer it is applied (layer transparency).

In order to support deep instantiated attribute [Atkinson and Kühne, 2001], an extension of the attribute algebra (Enhanced N-layer Attribute Algebra) has been given. The instantiation transformation has also been extended. Furthermore, it has been shown that if one has a valid
metamodel, then it is possible to instantiate it, refine the attributes on the model level, then re-instantiate the models and continue this process in an arbitrary depth. Moreover, the instantiation method always terminates and it produces a valid model for all possible metamodels.

The second construct presented in the chapter focused on defining the appearance of visual languages. A special visual language – supporting the description of appearance definitions – has been created. A model transformation method has been introduced that binds the appearance defined by this special language to the topology of the modeled language defined by the metamodel. Moreover, it has been shown that the transformation always terminates.

The constructs presented in the chapter solve two important shortcomings of metamodeling. Based on the presented attribute structure and on the appearance definition language, the construction of visual languages is much more flexible and efficient than before. The application-based validation of this statement can be found in Chapter 7.
Chapter 5

Optimized Metamodel-Based Constraint Validation

The growing importance of modeling and model transformation has attracted attention to create precise models and transformations. Visual model definitions have a tendency to be incomplete, or imprecise, thus, the definitions are often extended by textual constraints attached to the model items. These textual constraints can eliminate the incompleteness stemming from the limitations of the visual definitions.

Object Constraint Language (OCL) 2.0 [Warmer and Kleppe, 2003] is a formal language that remains easy to read and write. It is a subset of the OMG standard Unified Modeling Language (UML) [UML 2.0, 2006] that allows software developers writing textual constraints and queries over object models. A constraint is a restriction on one or more values of an object-oriented model or system. OCL is a useful approach to create precise UML models, but this preciseness is needed in the field of metamodeling and in the field of visual language construction as well. Although the original definition of OCL is limited to UML diagrams, the definition can be extended to support metamodeling. The main issue of the extension is to support a different, more generic modeling structure, than that specified by the UML Infrastructure. Furthermore, in order to preserve the completeness and preciseness of OCL, the formal specification of the language has to be extended, or a new formalism is required.

Using OCL in modeling made important to have efficient constraint handling frameworks. Supporting OCL requires an interpreter or a constraint compiler. Since modeling constraints are evaluated often, but changed rarely, using a compiler can improve the performance. Efficiency in OCL compilers can be reached mainly by optimizing the execution of the constraints. The performance of constraint evaluation is determined by two factors: (i) selecting the model items and their attributes that we need to check against the constraint, and (ii) executing the validation method. Usually, the first of the factors requires more time.

This chapter presents methods required by applying metamodel-based, optimizing constraint validation based on OCL. Section 5.1 introduces the existing approaches in the field of OCL handling, evaluation, extending OCL to support metamodeling, and OCL optimization. Section 5.2.1 discusses the extension of OCL to support the n-layer metamodeling infrastructure. This section presents a new formalism for OCL based on Abstract State Machines [Böerger and Stärk,
2003] as well. This formalism – referred to as OCLASM – can describe both the extended version of OCL and dynamic behavior, namely the evaluation of the constraints. Section 5.3 elaborates two optimization algorithms for OCL. The formal specification of the algorithms and the formal proofs of correctness for the algorithms are discussed as well. Finally, a summary is given about the discussed results.

5.1 Backgrounds and Related Work

5.1.1 Theoretical Results on OCL Formalization

OCL is a formal language, its formalization is based on set theory [OCL Spec., 2003]. However, the existing formalism does not cover all features of OCL [Flake, 2004] and it does not define the dynamic behavior of constraints. Furthermore, set theory is a flexible technique, but it uses a low-level description of the problem space and it does not support modularization.

The book [Richters, 2001] presents a precise approach, which facilitates the analysis and validation of UML models and OCL constraints. It defines a formal syntax and semantics of OCL types, operations, expressions, invariants, and pre-/postconditions, and it discusses some of the main problems with the original OCL specification. The book gives a precise overview about the background of OCL-based validation.

The OCL formalism available in set theory is examined in [Flake, 2004]. The paper collects the elements appearing in the OCL standard, but not in the formalism. It presents an extension of the original formalism to solve these problems.

In [Flake and Müller, 2004] an ASM definition for dynamic OCL semantics is presented. This formalism focuses on the states of the modeling environment and handles the invariants as atomic units implemented in outer functions, more precisely, it uses the original, set theory-based OCL standard to describe the constraint expressions. This means, however, that the formalism does not define the OCL expression evaluating mechanisms, instead it observes the execution on a high abstraction level.

Instead of extending one of the existing OCL formalization approaches, we have decided to create a completely new formalism. There were several reasons for this decision. The original formalism does not define the dynamic behavior of constraints, e.g. how the constraint expressions are evaluated. Since set theory is a highly flexible formalism technique, the formalism could be extended to support dynamic behavior. Abstract State Machines (ASMs) are working on abstract data structures, which are provided with a simple mathematical foundation. The notion of ASM is familiar from programming practice. ASM provides a concise way to define system semantics and dynamic behavior. The ASM formalism has several advantages in contrast with the extension of the original formalism in this field. Firstly, the notation of ASM is easier to use for proving the correctness of algorithms given by pseudo code. Secondly, modularization and stepwise refinement is easier to accomplish in ASM. This also means that the formalism specification of the dynamic behavior can be hierarchically decomposed. Set theory is a flexible technique, but it uses a low-level description of the problem space, thus, the description the dynamic behavior would produce a considerably huge rule set.
Still, we could have extended the formalization described in [Flake and Müller, 2004] to support our attribute structure. We have found however that the abstraction level used in the approach is not the abstraction level suitable for our purposes. The main problem was that in the approach, the dynamic behavior of the constraints are described by system states, where a system state is a snapshot of the running system e.g. a set of currently existing objects, a set of attribute values of the objects and a set of currently established links that connect the objects. However, the constraint itself does not appear directly in the machines of the formalism. Instead, the constraint expressions are applied indirectly, in one step by using the original, set theoretical formalism of OCL. This mechanism is not acceptable if we use the formalism to describe algorithms – e.g. optimizing algorithms – that can modify the constraint definitions. We need to use a higher abstraction level and define the constraint expressions by the formalization.

5.1.2 Tool Support for OCL

The Dresden OCL Toolkit (DOT) [Hamie et al., 1998] [Dresden OCL Toolkit, 2007] generates Java code from OCL expressions and instruments the system in five steps. (i) OCL expressions are parsed using a LALR(1) parser generated with SableCC [SableCC, 2007]. The result of the step is an Abstract Syntax Tree (AST). (ii) A limited semantic analysis is performed on the AST to find errors. (iii) The AST is simplified in order to make further processing simpler. (iv) The code generator traverses the simplified AST and builds Java expressions. (v) The generated code is inserted into the system that contains the constraint source code, thus, the contracts can be tested at runtime. Although DOT implements a real OCL compiler, it does not support metamodeling or optimization.

GME [GME Users Manual, 2000] supports constraint handling, it has a constraint interpreter called Constraint Manager. GME supports MultiGraph Constraint Language (MCL) that is a special dialect of OCL.

PROGRES [Schürr et al., 1999] is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. PROGRES supports pre- and postconditions. The precondition of a transaction is a query, which should never fail when applied to the input graph of the surrounding transaction. Similarly, the postcondition of a transaction is a query, which should never fail applied to the output graph of the surrounding transaction. It is allowed to access the in- and out-parameters of the transaction, but it does not distinguish between a before- and an after-state of referenced nodes.

VIATRA [Varró and Pataricza, 2003] is a model transformation framework developed mainly for the formal dependability analysis of UML models. In VIATRA model constraints are graph patterns with arbitrary levels of negation.

Kent Modeling Framework [Akehurst and Patrascoiu, 2004] is a set of tools to support model driven software development. One of these tools is KMFStudio, a tool to generate modeling tools from metamodels. KMFStudio supports dynamic evaluation of OCL constraints. It enables the language to be bridged to other modeling frameworks. The tool was integrated into the Eclipse tool set. The Kent Modeling Framework does not use optimizing algorithms to improve the efficiency of the constraint validation.
Open Source Library for OCL (OSLO) [Open Source Library for OCL, 2007] is a new tool, it is a further development of the Kent OCL Library. OSLO is based on the Eclipse framework. OSLO supports OCL 2.0 functionality for arbitrary metamodels based on EMF, and constraint checking for UML2 models (Eclipse Uml2). OSLO supports constraint checking for metamodeling systems. Since it is a recent project, only a few publications are available, and not all of the supported features are introduced in depth.

Object Constraint Language Environment [Object Constraint Language Environment, 2007] is a UML CASE Tool. The tool helps the users to apply both static and dynamic checking at the user model level. The tool also has a user-friendly graphical GUI. Although the tool supports model checking, it does not use compiling techniques, or constraint optimization.

This thesis elaborates two optimization algorithms for OCL. The basic idea of the first algorithm, RelocateConstraint, is borrowed from [Lengyel, 2006]. The mentioned work introduced a constraint relocation algorithm NormalizeConstraint to optimize the number of model queries in an aspect-oriented constraint weaving approach. Although the basic idea is very similar to that of RelocateConstraint, there are differences. First of all, in [Lengyel, 2006], constraints are evaluated for previously selected (matched) model items, not to all model items. Thus, we can construct a model item registry that stores which model item has been relocated in which model item. In the general case, such as in the optimization method presented in this chapter, we can map only the original and the new context (the metamodel items) to each other, but not the model items of the context (the instantiation model items). This is the reason for example why the DecomposeConstraint algorithm (focusing on decomposing the constraints to AND/OR clauses and apply these clauses separately), can be applied there, but cannot be used here. Another difference is that NormalizeConstraint algorithm is not intended to support handling multiplicities. However, the complexity of the presented optimization algorithm comes exactly from handling all possible multiplicity combinations. Therefore, the presented algorithm not only mirrors the ideas, but extends, improves and generalizes them.

5.2 Constraint Validation

5.2.1 Extending OCL

The OCL specification is based on the attribute structure defined in UML. However, this attribute structure does not support layer-transparency as described in Chapter 4. In the followings, we introduce an extension for OCL, which supports n-layer metamodeling. Note that the approach has borrowed the basic idea of formalization – namely handling the constraint code as annotated syntax tree – from [Stärk et al., 2001].

5.2.1.1 Supporting the N-layer Attribute Algebra

N-layer Attribute Algebras represent the attributes of model items in a much more generic way, than UML. Although the attribute structure of both N-layer Attribute Algebras contains hard-coded elements (InstanceName, Attribute and ComplexType), the structure is not restricted
to these elements. The algebra allows defining new attributes with customized structure by modifying the metamodel. By using attribute triples, it supports deep-instantiation as well.

**Example 5.1.** There is a visual language to represent a computer network. The nodes of the network are desktop computers with different characteristics described by three attributes: ComputingCapability, OperatingSystem and Owner. ComputingCapability shows the relative performance of the node by using a number between 1 and 1000. OperatingSystem is a string attribute, while Owner defines the Name, EmailAddress and PhoneNumber of the owner of the computer. In this case, the structure of Owner (Name/EmailAddress/PhoneNumber) is defined by the metamodel, it is not hard-coded in the algebra.

The extension of OCL must support traversing the metamodel-based attribute structures including complex attributes with subattributes. In order to achieve this, we define exactly how these subattributes can be referenced: our approach uses a new type of OCL expressions: the `AttributeNavigation`. The notation of `AttributeNavigation` is the same as the notation of navigation between model items in the standard OCL: steps of the navigation are separated by dots, if the destination is not uniquely determined by the name used in the expression, then the result is a set. For example, the expression `Owner.Name` represents the Name subattribute of Owner. If an owner can have more than one email address, then `Owner.EmailAddress` obtains the set of all email addresses of the owner.

Note that using the same notation for `AttributeNavigation` and navigation steps between the model items does not lead to inconsistency, because normal navigation steps are not allowed after the first `AttributeNavigation` since we cannot navigate from an attribute to a model item.

The second and third components of the attribute triples (namely the Boolean field showing whether re-instantiation is required and the string field specifying the re-instantiated name of the attribute) do not appear at this level, because these properties are rather instantiation-specific, than attribute-specific.

### 5.2.1.2 Supporting Attributed Relations

Most of the UML models, such as class diagram, or object diagram handle attributes only in modeling types, not in relations between the types. Instead of attributed relations, a UML class diagram uses association classes to describe the attributes of a relation. This restriction, namely, that the relations cannot have attributes is inherited by OCL. More precisely, according to the OCL specification, constraints can be created for nodes and association nodes, but the edges are used only to support navigation between the nodes. In order to support attributed edges (which is a common need in metamodeling), this limitation must be resolved.

The main idea behind the extension of OCL is that the edge attributes can be modeled as an edge without attributes and association class that holds the attributes. This solution does not require any modification in the OCL grammar, but the compiler or interpreter evaluating the constraints has to be extended accordingly.
5.2.2 OCLASM: A New Formalism for OCL

OCLASM has been created to formalize our OCL dialect and the evaluation of constraints. OCLASM acts as an interpreter for OCL constraints, it describes the constraint expressions. The same functions and the same rule set are used regardless of the constraint or the underlying model. States of OCLASM represent the state of execution at a certain point of time. A state can be considered as an internal state of the evaluating environment, which is evolved by the rule set of OCLASM. States describe for example which expression is under evaluation or which local variables are available. As usual in case of ASM, a state is considered as a configuration for functions of OCLASM. Functions in this case are used to query the underlying model, obtain the expressions from the constraint and to represent the internal state of the interpreter simulated by OCLASM. Note that this also means that the initial state, namely the initial values of functions may differ.

The rules of OCLASM are used to navigate between the states when running the validation. OCL cannot change the underlying model by the definition [Warmer and Kleppe, 2003], thus, model queries are formalized as monitored functions. Similarly, the expressions of the constraints are obtained by a monitored function. The execution of the constraint is a step-by-step execution of these programming units using the rules of OCLASM.

Monitored and Shared Functions

Recall that obtaining the model items and the phrases of constraint expressions is handled by monitored functions. This solution ensures that the modeling environment and the evaluation environment are independent from the dynamic behavior of the constraint formalized by the OCLASM. The underlying model extends the original model structure definition used in OCL: it consists of model nodes, attributes, and relationships not restricted to UML types. Attributes are either of primitive types or of complex attributes containing several sub-attributes. Complex attributes can be nested hierarchically. Model nodes can contain attributes: both primitive and complex attributes. The attribute and modeling structure described in Chapter 4 is used, which extends the UML-based modeling structure to an n-level metamodeling hierarchy. The extension of the underlying modeling structure means that OCLASM can be used not only to describe the dynamics of constraint evaluation, but to formalize constraints for domain-specific models as well.

Both model nodes and attributes are identified by a unique ID (a literal expression), the universe of IDs is shared between the two different constructs. This uniformity helps reducing the number of monitored functions (for example, node-node and node-attribute navigations can be handled uniformly). To differentiate attributes and normal model nodes, there is a IsModelNode(ID) function defined. To simplify attribute handling, the function AttrValue(ID) is used. The function returns undef for complex attributes, and returns a primitive value for primitive attributes. A primitive value in this context means that the type of the value is a Boolean, a number (integer or float) or a string.

To express the meta level – instance level relationship, the Meta(ID) function is used, which returns the meta item of the instance level item identified by ID.
The navigation between model items is handled by two functions. The function `To(ID, Dest)` works on the model level, parameter `ID` identifies the source model item, while `Dest` selects target items. The function returns all nodes or attributes which can be reached from the model item using a relation where the destination name is `Dest`. The result of the function is a list of possible destinations. Note that the list contains either model item `IDs`, or attribute `IDs`, but not both of them, because OCL does not allow this kind of polymorphism. The `Mul(ID, Dest)` ("Mul" stands for Multiplicity) function is another function to handle relations between model items. It works on the metamodel level, thus, the model definition is checked instead of the concrete models (the type definition instead of the instance model). The function `Mul` checks the minimum and the maximum multiplicity of the given relation according to the metamodel. The multiplicity of relations in metamodels is defined by a range of possible multiplicities. Usually, this range is defined by two integers representing the lower, and the upper limit. However, there are other ways to configure valid multiplicities, e.g. by enumerating on all valid values. The function returns four integers as a list that are the minimum/maximum multiplicity (`inf(R)`, `sup(R)`) of the source/destination side.

OCLASM uses a shared function `GetPhrase(Position)` as well. `Phrases` are basic syntactic constructs (programming statements and expressions) available in OCL. A `Phrase` has a string attribute `PhraseType` (e.g. "NavigationCall"). `Phrases` can contain other `Phrases` as children. For example, an iteration `Phrase` can have an iterator variable declaration `Phrase`, an iteration condition `Phrase` and an iteration core block `Phrase`. The function `GetPhrase` returns a `Phrase` of the constraint identified by the parameter `Position`. `Position` is a value from the universe of all possible positions of `Phrases` of the constraints (the universe is referred to as `UPos`, while positions are referred to as `tree positions` in order to distinguish them from navigation positions of model queries). Since the constraint is handled as a syntax tree during the evaluation, thus, `Phrases` are the nodes of this tree and the universe of `UPos` contains the pointers to the nodes. Note that the function is marked as shared, which means that it can be updated by the environment, or by the rules of OCLASM. This duality is required, because in general, constraints are defined outside the scope of OCLASM, but certain algorithms can modify the original constraints. For example, an optimization algorithm can restructure the expressions of constraints in order to improve the performance.

`Child(Position, I)` is another shared function. It obtains the `I`th children `Phrase` for a `Phrase` that has at least one child. The first parameter of the function is the (valid) tree position of the complex `Phrase`. If the `Phrase` does not have a child at the selected index, the function returns `undef`. The function `Parent(Position)` implements the reverse direction: it obtains the tree position of a `Phrase` and returns the tree position of the container `Phrase`. These functions are shared functions for the same reason as `GetPhrase`. Note that `Child` and `Parent` functions are always synchronized automatically by the executing environment.

**Dynamic Functions**

The constructs of OCL, for example iterate or navigate, are mainly defined as rules in OCLASM. Dynamic functions help to store the current state of the evaluation environment when the rules are applied. Dynamic functions are similar to helper variables in the environment framework.
The nullary function \textit{SelfReference} is used to retrieve the context of the constraint. The value of the function is set, when a new constraint is evaluated.

To obtain the current tree position of evaluation, i.e. the tree position of the \textit{Phrase} currently under execution, the nullary function \textit{CurrentPos} is used. The return value of the function is a tree position from the universe \(U_{\text{Pos}}\).

The dynamic function \textit{Type(}\textit{Pos)}\textit{)} is used to handle the type of the different (OCL) expressions uniformly. Its parameter is the tree position of the target expression. The return value of the function can be one of the basic types defined in OCL such as \textit{Boolean}, \textit{real}, \textit{integer}, or \textit{tuple}. The value of the expressions are handled similarly to the type function: the unary function \textit{Value(}\textit{Pos)}\textit{)} retrieves the tree position of the expression and returns its value. Using the notation of common programming languages, such as C, the difference between \textit{GetPhrase(}\textit{pos)}\textit{)} and \textit{Value(}\textit{pos)}\textit{)} is that \textit{GetPhrase(}\textit{pos)}\textit{)} is similar to a pointer. In contrast, \textit{Value(}\textit{pos)}\textit{)} is the value in the pointed memory block. OCLASM allows updating the \textit{Type/Value} functions, because the value of these functions is not set before the appropriate constraint segment has been evaluated (we do not know the result of an operation before we apply it).

OCL allows the user defining local variables. In OCLASM, these variables are handled by the function \textit{Local(}\textit{Name)}\textit{)}. The function has one input parameter: the name of the variable. The function \textit{Local} returns the tree position of a variable declaration expression. Variable declarations contain the name, type and value of the variable. If a local variable is requested by its name, and there is no local variable defined with the given name, then the function returns \textit{undef}. The name of the local variables is handled by the unary function \textit{Name}, which has one input parameter: the tree position of the variable expression. \textit{Name} returns the name of the local variable as a string, or it returns \textit{undef} if the tree position is not a valid variable definition.

OCLASM handles all four types of collections (\textit{Set, OrderedSet, Bag} and \textit{Sequence}) by arrays indexed by integer numbers. Indexing is denoted by brackets. The items in the arrays are the items in the collections, for example, the expression \textit{Value(}\textit{Position)}\textit{[3]}\textit{)} means the third item in the collection expression at the tree position \textit{Position}. The arrays can be traversed by the \textit{forall} expression of ASM, obtaining every element. According to the default notation of ASM [Böerger and Stärk, 2003] the length of collection is denoted as \textit{l(Value(}\textit{Position)}\textit{))}, where \textit{Position} is the tree position of the expression.

Tuple types are also handled as arrays indexed by integer values, the definition of tuple items are stored in lists with two elements (name - value pairs), these lists are composed from the list items of the array representing the tuple. For example, the expression \textit{Tuple(x:Integer = 5, y: String = 'Ok')} results an array with two items: TupleArray[1] = ['x', Integer = 5] and TupleArray[2] = ['y', String = 'Ok']. When a tuple item is queried by its name, then OCLASM tries to find an item in the associated array with the name and updates the \textit{Type/Value} functions. In the previous example, if the tuple item with name 'y' is requested, OCLASM checks TupleArray[1], but its name ('x') does not match, thus, it advances to TupleArray[2]. Since the name is found, OCLASM sets the \textit{Type} of the current tree position to 'String' and the \textit{Value} to 'Ok'.


Vocabulary and Universes

Definition 5.2. The superuniverse $|\mathfrak{A}|$ of a state $\mathfrak{A}$ of OCLASM is the union of six universes:

- (i) The universe of *Phrases* (basic syntactic constructs of OCL as defined in the EBNF definition of OCL [Warmer and Kleppe, 2003]) ($U_{\text{Phrases}}$)
- (ii) The universe of possible tree positions of *Phrases* in the constraints ($U_{\text{Pos}}$)
- (iii) The universe Boolean (true/false/undef, $U_{\text{Bool}}$)
- (iv) The universe of finite lists of numbers ($U_{\text{Number}}$)
- (v) The universe of finite lists of finite strings ($U_{\text{String}}$)
- (vi) The universe of finite lists of possible identifiers (IDs) for model items and attributes ($U_{\text{ID}}$).

In order to define the vocabulary of OCLASM as well, firstly we describe the model and constraint structure. The underlying model of OCLASM is defined as a Labeled Directed Graph (see Section 3.3 for definition). For sake of simplicity, we handle labeling functions $l_{\text{e}}^G$ and $l_{\text{i}}^G$ by a dual field notation representing *Name / Value* pairs. We refer to a label with the name $N$ of the model item $X$ as $X_N$. Attributes are modeled as nodes, not as labels. We define special labels as follows:

- For each node and edge $X$ we define
  - a globally unique label $X_{ID}$ that takes its value from $U_{ID}$.
  - a label $X_{ModelItem}$ that takes its value from $U_{\text{Bool}}$.
  - a label $X_{TypeOf}$ that takes its value from $U_{ID}$.
  - a special, complex label $X_{\text{ContainedAttributes}}$ that contains a possibly empty list of $U_{ID}$s.
  - a label $X_{Value}$. The label returns *undef* if the node is not an attribute with simple type, otherwise it contains the value taken from $U_{\text{Bool}}$, $U_{\text{Number}}$, $U_{\text{String}}$, or $U_{ID}$.

- For each edge, $X$ we define
  - labels $X_{\text{SrcName}}$ and $X_{\text{DestName}}$ that takes their value from $U_{\text{String}}$.
  - labels $X_{\text{SrcMinMul}}$, $X_{\text{SrcMaxMul}}$, $X_{\text{DestMinMul}}$ and $X_{\text{DestMaxMul}}$ that takes their value from $U_{\text{Number}}$.

The monitored functions of OCLASM are defined as follows:
We create another special labeled, directed graph to describe the structure of constraint expressions. Nodes represent the *Phrases* of the constraint, while contained-container relationship is modeled by edges between the nodes. These directed edges start from the container and points to the contained *Phrase*. We define two labels only:

- For each node \( N \), we define a globally unique label \( N_{Position} \) that takes its value from \( U_{Pos} \).
- For each edge \( E \) we define a label \( E_{ChildPosition} \) that takes its value from \( U_{Number} \).

Using these functions, we can define the vocabulary of OCLASM:

**Definition 5.3.** The vocabulary OCLASM of the OCLASM formalism is assumed to contain the following characteristic functions (arities are denoted by dashes):

**Monitored functions:** IsModelItem/1, AttrValue/1, Meta/1, To/2, Mul/2

**Shared functions:** GetPhrase/1, Child/2, Parent/1

**Dynamic functions:** SelfReference/0, CurrentPos/0, Type/1, Value/1, Name/1, Local/1
Transition Rules

Transition rules describe how the states of OCLASM change over time by evaluating expressions and executing statements of the input program. The main idea is to create a rule for each type of language constructs available in OCL. For example, OCLASM has rules for iterate, navigation, or variable declaration actions. These rules describe the semantics of the expression and manage dynamic functions. OCLASM has a central rule called `eval`. The rule `eval` has a switch-case block with many branches, more precisely for each type of language constructs, `eval` has a branch. The rule checks the type of the parameter `Phrase` and executes the appropriate branch by calling the rule associated with the `PhraseType`. Therefore, `eval` acts as a mapping function between `Phrases` and rules of OCLASM. Moreover, `eval` helps in calling the rules with the appropriate parameters (it adds sub-expressions as parameters, using the Child function). Updating the value of `CurrentPos` is also handled by `eval`.

Although this rule could be removed from OCLASM and the logic coded in the rule could appear directly as a set of rules of OCLASM (as in [Stärk et al., 2001]), we have decided to create `eval` in order to simplify handling of rules implementing language constructs.

The initial position of OCLASM sets the `CurrentPos` to the start tree position of the outermost constraint expression and sets the value of all other dynamic functions to `undef`. A run of OCLASM is started by calling `eval` for the start tree position. When the run of the state machine of OCLASM is finished, the outermost expression holds a single value. If the evaluated constraint was an invariant, then this value shows whether the model was valid.

Invariants

OCLASM, as presented until this point is useful to simulate the execution of a simple or complex OCL expression, but not a whole constraint, such as an invariant. The current description shows how invariants can be formalized. Note that the `forall` expressions are executed sequentially in order to use `CurrentPos` property correctly.

**Algorithm 5.1 The CheckModelInvariants rule**

```plaintext
1: rule CheckModelInvariants(Invariants)
2: for all Invariant in Invariants do
3:   for all ID ∈ {MODELITEM(ID) = true} do
4:     if Invariant.Context = META(ID) then
5:       SelfReference := ID
6:       CurrentPos := Invariant.StartPos
7:     LOCAL := undef
8:     if not eval(CurrentPos) then
9:       INVALIDMODEL
```

To have this rule as part of OCLASM, the universe of invariants must be added to the superuniverse in 5.2. Invariants have a `Context` property and a reference to the outermost expression in the invariant. Pre and post conditions can be handled similarly by extending the superuniverse of OCLASM and specifying the execution semantics of the constructs.
Rule Implementations

Firstly, a very simple rule, the _VariableDeclaration_ is shown. The function _eval_ obtains the tree position of the children expressions (variable name, type and initialization value) and computes them before this rule is executed.

```plaintext
rule VariableDeclaration(VarName, VarType, VarInit)
    Name(CurrentPos):= Value(VarName)
    Type(CurrentPos):= Value(VarType)
    if (VarInit!= undef)
        Value(CurrentPos):= VarInit
    else
        Value(CurrentPos):= undef
```

Secondly, the rule for iterate operations is presented. It is essential to formalize this operation, because every other collection operation can be accomplished by using iterate [Warner and Kleppe, 2003]. For example, the collection operation `count()` can be simulated by an iterate expression `iterate(i : Integer, r Integer = 0 | r+1)`. The node iterate has exactly four children in the abstract syntax tree: (i) the collection where the operation is applied; (ii) the declaration of the iterate variable, (iii) the declaration of the result variable, and (iv) the iteration body.

```plaintext
rule iterate(CollectionDef, IteratorDecl, ResultDecl, Iteration)
    Local(Name(ResultDecl)):= new(Pos)
    Value(Local(Name(ResultDecl))):= Value(ResultDecl)
    Type(Local(Name(ResultDecl))):= Type(ResultDecl)
    forall collectionElement in Value(CollectionDef)
        Local(Name(IteratorDecl)):= new(Pos)
        Value(Local(Name(IteratorDecl))):= collectionElement
        Type(Local(Name(IteratorDecl))):= Type(IteratorDecl)
        eval(Iteration);
    endfor
    Local(Name(IteratorDecl)):= undef
    Value(CurrentPos):= Value(Local(Name(ResultDecl)))
    Type(CurrentPos):= Type(Local(Name(ResultDecl)))
    Local(Name(ResultDecl)):= undef
```

The third example shows the rule constructed for navigation expressions. Here the model-based, external functions are also used. The rule evaluates the origin, namely, it obtains the model item which is the starting point of the navigation. As next, the rule checks the multiplicity of the rule, if it allows exactly one connection, the result is a _ModelItem_, in any other case the result is a collection of _ModelItems_. Since the function _To_ returns always a list (with the _IDs_ of the destination nodes), in the first case the first element of the result array is used (To(Value(Origin), DestName)[0]). In this case the type of the result is the ID of the meta node of the destinations node. If the multiplicity is not 1, then a new collection is created and returned.
rule Navigate(Origin, DestName)
    if (Mul(Value(Origin),DestName)[2]=1 and
        Mul(Value(Origin),DestName)[3]=1 and
        IsModelNode(To(Value(Origin),DestName)[0]))
        Value(CurrentPos):= To(Value(Origin),DestName)[0]
        Type(CurrentPos):= ModelItem
    else
        Type(CurrentPos):= Set
        Value(CurrentPos):= Set()
        forall ModelId in To(Value(Origin),DestName)
            Append ModelId in Value(CurrentPos)
        endfor
endif

5.3 Constraint Optimization

It is essential to remain unchanged the result of the constraint evaluation by the optimization algorithms. A constraint modification is said to be correct if, and only if the output of the optimized and original constraint is the same for every possible input. In general, correctness is even more important, than efficiency. Thus, it is rigidly checked whether the presented algorithms are correct.

In the following section, two optimization algorithms are presented. In the optimization algorithms we assume that the model does not change under evaluation because of external systems, or users. More precisely we assume that the evaluation is aborted and restarted if the model has changed. The first algorithm tries to relocate the constraint to a new context, where the evaluation requires less model queries. The second algorithm realizes a completely new idea, it is an efficient caching method that is based on the specialties of OCL. Note that the optimization algorithms support the optimization of invariants only, which is possibly the most important subset of OCL constraint types. Extending the optimization to other constraint types is part of our current research (see Section 8.3).

5.3.1 Constraint Relocation

One of the most efficient way to accelerate the constraint evaluation is to reduce the navigation steps in a constraint without changing the result of the constraint. This is the aim of the first algorithm called RelocateConstraint. The algorithm tries to find the ‘optimal’ model item for the constraint, where optimal means that the evaluation of the constraint needs the least model queries possible. However, the relocation is not always applicable, there are several restrictions on the algorithm. Firstly, we introduce these restrictions, secondly, we present the algorithm itself.

5.3.1.1 Restrictions

Since the original and the optimal node are not always neighbors, the optimization stores a path between the original and the new context. This acyclic, finite path is called RelocationPath. Note
that cyclic paths would lead to a loop executable infinite times resulting negative or positive infinite navigation cost. In order to avoid this, we store previously visited model items when constructing the path and avoid edges, which would form a loop.

Selecting the exact path between the original and the optimal node is necessary, because there can exist more than one paths between the two nodes in the host graph. The differences between the paths can mean that one path is acceptable, while the other is not. Where an acceptable RelocationPath means a path that results in a correct relocation of the constraint.

The aim of the limitations is to eliminate the cases where the result of the original and the optimized algorithms would differ. To achieve this, it is necessary to examine when and how correct relocations can be applied. In the following propositions, we often say — for the sake of simplicity — that a RelocationPath is correct, although we mean that the relocation using the RelocationPath is correct.

**Proposition 5.4.** If the steps of RelocationPath are separately correct, then their composition, the RelocationPath is also correct.

**Example 5.5.** The original constraint is located in node A, the optimal node is D (Fig. 5.1). Thus, the RelocationPath is drawn from A to D (dashed line). If neither the relocation from node A to C (solid line), nor the relocation from node C to D (dotted line) change the result of the constraint, namely they are correct, then the proposition states that the relocation from A to D is also correct.

![Fig. 5.1. The steps and the whole RelocationPath](image)

**Proof.** Let \( C \) be the original constraint and \( P \) a complex RelocationPath found by the search steps. \( P \) contains finite number of steps by definition.

Furthermore, let \( O \) be the original context; \( S \) the first step of \( P \) and \( O' \) the destination node of \( S \) in \( P \). According to the premise of the proposition, the correctness of \( S \) is proven, thus, relocating the constraint from \( O \) to \( O' \) can be accomplished. After applying this relocation, a new constraint, \( C' \) can be constructed. Applying the relocation algorithm on \( C' \) results a new RelocationPath, \( P' \) containing one less step, than the original one. Since \( P \) has a finite number of steps, the algorithm always terminates.

**Corollary 1.** The steps in a path can be examined separately. If in a certain case the correctness of the algorithm is proven for each single navigation step in the RelocationPath, then it is also proven for the whole RelocationPath. Thus, in general, if the correctness of each possible single navigation step is proven, then the correctness of the whole relocation path is proven as well. Therefore, it is enough to examine the correctness of the individual relocation steps.

In the next propositions, the following abbreviations are used: \( C \) denotes the original constraint, \( C' \) the new (relocated) constraint, \( M_0 \) is a metamodel, \( M \) is a model (an instantiation
of $M_0$), $O$ is the original context, $N$ is the new context. $O$ and $N$ are metamodel elements, and their instantiations are $O_1, O_2, \ldots, O_n$, and $N_1, N_2, \ldots, N_n$ respectively.

**Example 5.6.** Fig. 5.2 shows an example metamodel, its instantiation, and the constraint relocation. The metamodel represents a domain that can model computers and display devices (here monitors only). A single computer can use multiple monitors. The model defines a simple constraint attached to the node **Computer**, this constraint is relocated by the optimization to the node **Monitor**.

Using the abbreviations, we can say the following: $M_0$ is the metamodel shown in Fig. 5.2/a, $M$ is its instantiation (Fig. 5.2/b). $O$ corresponds to **Computer**, $N$ corresponds to **Monitor** in $M_0$. $O$ has two instantiations, **Computer1** ($O_1$) and **Computer2** ($O_2$). Similarly, PrimaryMonitor is $N_1$, SecondaryMonitor corresponds to $N_2$, and **Monitor** to $N_3$.

Recall that the multiplicity of the relationships in metamodels is defined by a range $R$ of possible multiplicities. A multiplicity allowing zero value means that there can be unconnected nodes in the relation. A multiplicity allowing a value greater than one means that navigation between the nodes must use a set operation. Relations need different handling according to their multiplicities, thus we distinguish three categories:

- **ZeroOrOne** - $inf(R) = 0 \land sup(R) = 1$
- **ZeroOrMore** - $inf(R) = 0 \land sup(R) > 1$
- **ExactlyOne** - $inf(R) = 1 \land sup(R) = 1$
- **OneOrMore** - $inf(R) = 1 \land sup(R) > 1$

Multiplicities on the source and on the destination side have different meanings, thus, possible modeling structures can be described by a combination of the presented multiplicity categories. For example, in the previous metamodel (Fig. 5.2) the multiplicity is **ExactlyOne** (source side) - **OneOrMore** (destination side).
The following propositions are based on the multiplicity combinations. This first case is the most simple, allowing only *ExactlyOne* multiplicity on both sides.

**Proposition 5.7** (Case A). A relation with multiplicity *ExactlyOne* on both sides can be used for relocation. In this case the relocated expression differs from the original version in the navigation steps (or navigation step sequences) only. The new constraint expression is transformed from the original definition using the following rules:

**Rule 1.** If the expression is a navigation to the new context (N), then the expression is transformed into *self*.

**Rule 2.** If the expression is an attribute query in the old context (O), then the new expression is a navigation from N to O and an attribute query applied there (e.g. *self.Manufacturer* is transformed to *self.computer.Manufacturer*).

**Rule 3.** If the expression is a navigation from the old context (O), then the new expression is a navigation from N to O.

**Rule 4.** Other expressions in the constraint are not altered.

**Example 5.8.** Let the example metamodel presented above define that computers are able to handle exactly one monitor, and monitors are always connected to exactly one computer (Fig. 5.3). Furthermore, let the constraint $C$ state that the monitor is an LCD monitor (display.Type $= \textquoteleft\text{LCD}\textquoteright$). In this case relocating the constraint will result $C'$: Type $= \textquoteleft\text{LCD}\textquoteright$.

![Fig. 5.3. *ExactlyOne* multiplicity on both sides - metamodel and model](image-url)

**Proof.** An *ExactlyOne* multiplicity on both sides means that O and N objects can refer to each other the same way (using the role name of the destination node). The result of the navigation reference is always a single model item, not a set of model items and not an undefined value. This means that changing the navigation steps can be accomplished.

The transformation rules remains correct if the rules above are satisfied:

**Rule 1.** The relocation has changed the context, thus, the navigation step in the original context is not necessary any more.
Rule 2. and Rule 3. Since the original attribute reference, or the destination node of the navigation is invalid in the new context, thus, the constraint has to navigate back to the original context first, and apply the expression there.

Rule 4. Rule 1-3. covers all possible valid attribute and navigation expressions, thus, no additional rules are required.

Handling multiplicity different than *ExactlyOne* on the source side can be applied by rewriting the constraints:

Proposition 5.9 (Case B). Navigation edges that allow zero multiplicity (*ZeroOrOne*, or *ZeroOrMore*) on the source side can be used in RelocationPath by encapsulating the original constraint by a `notEmpty` expression that checks whether the original context is accessible from the new one. If the original context cannot be reached from the new context, then the constraint results true showing that the expression cannot be evaluated (it does not violate the invariant).

Example 5.10. Fig. 5.4 shows an example metamodel and model for the proposition. Let \( C \) be defined as \( \text{self.monitor.Price} < 50 \). If this constraint is relocated, then it is transformed to
\[
\text{if (self.computer->notEmpty()) self.Price < 50 else true}
\]
expressing that each monitor has at least one computer attached, with a price less than 50$. The invariant is automatically evaluated to *true* in \( \text{Monitor1} \).

Proof. Let \( M \) be a model with \( O_1, N_1 \) and \( N_2 \) defined (Fig. 5.4). Let \( N_1 \) be isolated (or at least not connected with \( O_1 \)). If the source class is present, then the evaluation can be applied the same way as in Case A (Prop. 5.7). This is the case in \( N_2 \). If the source class is not present, then the original constraint does not check the destination class (as in \( N_2 \)), thus the class is not required to check by the relocated constraints either. The encapsulating `notEmpty` expression ensures that only those nodes are checked, where the source class is presented. This means that the relocation is always *correct.*
Proposition 5.11 (Case C). If the multiplicity is ZeroOrMore, or OneOrMore on the source side, we collect the original context nodes in a set, and iterate on this set to check the relocated constraint in each nodes of the old context. Therefore, the constraint expression is relocated by adding an encapsulating `forall` expression to the constraint (similarly to `notEmpty` in Case B (Prop. 5.9)). The `forall` expression iterates on all of the original context nodes, where the iterator values are the original context nodes (see example below). This also means that inside the `forall` expression, this case is similar to Case A (Prop. 5.7). If the relocated constraint does not contain any attribute reference to the original context node, or navigation through it, then the `forall` expression can be avoided.

Example 5.12. Let O contain a simple constraint referring to one of its attributes, named `Abstract`. After the relocation, the constraint is located in N and the reference `self.Abstract` is transformed to

```
self.O->forall(O | O.Abstract).
```

This `forall` expression is true only if the condition holds for every elements in the set.

Example 5.13. The example model has been changed to meet the requirements of the proposition (Fig. 5.5). Let C be defined as `self.Price < display.Price`. If this constraint is relocated, then it is transformed to

```
self.computer->forall(computer | computer.Price > self.Price)
```

expressing that each computer attached to the monitor has to accomplish the condition. Note that the navigation from O to N in `display.Price` was reduced to a single `self` reference.

**Fig. 5.5.** MoreThanOne $\rightarrow$ ExactlyOne multiplicity - metamodel and model

Proof. The presented method ensures that each model item on the original source side is processed, and the constraint is checked for all of them. If the multiplicity on the source side is ZeroOrMore, then it is necessary to check first whether a node with the original context is reachable (using the constructs of Case B (Prop. 5.9)), and apply the `forall` expression only if it is reachable. This method ensures that the navigation back to the original node is always
possible. Therefore, inside the iteration, it is able to use the constructs of Case A (Prop. 5.7), since in each iteration step there is exactly one node examined from the original context. Inside the `forall` loop, the name of the destination node is the iterator value to make rewriting simpler. Thus, the relocated and the original version are always equivalent.

Multiplicities, other than `ExactlyOne` on the destination side is harder to handle, than on the source side. The following propositions describe when and how the relocation is possible in these cases:

**Proposition 5.14** (Case D). *Navigation edges that allow zero multiplicity (ZeroOrOne or ZeroOrMore) on the destination side cannot be used in RelocationPath.*

*Proof.* Since the metamodel allows zero multiplicity on the destination side, therefore, it is possible to construct a model, where the constraint disappears during relocation: Let $M_0$ allow zero multiplicity between $O$ and $N$ (on the destination side). This means that a model $M$ — which contains the instantiations of $O$ without a connected object of $N$ — is a valid model. If the constraints of $O$ are relocated to $N$, then the constraint is completely eliminated. Since it is always possible to construct this counterexample, relocation cannot be used in this case.

*Partial relocation* means that some of the expressions are executed in the new context, while others are executed in the original context. We use the term *semi-relocated* for expressions executed in the original context, and *fully relocated* for expressions executed in the new context.

**Proposition 5.15** (Semi-relocation). *If the constraint contains more than one attribute reference expressions, then partial relocation is always feasible. The original context is reached using navigation. Partial relocation cannot be applied if zero multiplicity is allowed on the destination side.*

*Proof.* Since the proposition is true only for relations not allowing zero multiplicity on the destination side, the navigation between the original and the new context is always possible. This means that all relations can be traversed according to the constructs presented earlier. Therefore, it is always possible to navigate back to the original context and evaluate the constraint there. In this way, the relocated and the original functionality is the same.

*Partial relocation* is useful, when the limitations of the relocation algorithm do not allow executing the whole constraint in the new context as in the following cases.

**Proposition 5.16** (Case E). *If the multiplicity is OneOrMore on the destination side, then only those constraint expressions can be fully relocated, where the original expression uses `forall`, or `not exists` to obtain the referenced model items of the new context. This means that only those expressions can be used here, which select all of the model items, or none of them (no partial selection or other set operation are allowed).*

*Example 5.17.* The expression `self.N->count()` or `self.N->select(N.IsUnique)` cannot be relocated, but the expression `self.N->forall(N.IsUnique)` can.

*Example 5.18.* The example model shows the requirements of the proposition (Fig. 5.6).
Note that due to the preconditions of the proposition, the references to Monitor are always set operations in Computer. For example, the expression self.display.Price > 300 cannot be used, because display is a set, not a single value.

Let \( M_0 \) contain three constraints in context Computer: \( C_1, C_2 \) and \( C_3 \) using the following definitions:

\[
\begin{align*}
\text{inv } c1 &: \text{ self.Price > 650} \\
\text{inv } c2 &: \text{ self.display->count() > 5} \\
\text{inv } c3 &: \text{ self.display->forall(m:Monitor| m.Price<300)}
\end{align*}
\]

The proposition requires the constraints to use forall operation to query the attributes of the new context, or the navigation paths through the new context. The proposition does not restrict other type of operations, which do not navigate to the new context (for example a local attribute queries, such as in \( c1 \)). In this case the rules of Case A (Prop. 5.7) can be used, thus, \( C_1' \) becomes the following:

\[
\text{inv } c1: \text{ self.computer.Price > 650.}
\]

Complex set operations (e.g. \( C_2 \)) cannot be fully relocated according to the proposition. This limitation does not apply to \( C_3 \):

\[
\text{inv } c3: \text{ self.Price<300.}
\]

Although the original and the relocated version of the constraint seems to differ, they have the same meaning: all monitors must be cheaper than 300 USD.

Proof. Firstly, we classify set operations into two categories, then we show that the statement of the proposition is true in both cases, in both categories.

The first category contains all operations, which use possibly more than one item of the set in the iteration. For example count is such an operation, since it uses the existence information of all items in the set. Obviously, such operations cannot be fully relocated, since the destination model items do not have any information about other model items in the set (other destinations of the original navigation).
The second category contains operations, in which the iteration steps are using always only one item of the set. This means, that the core of the operation can be evaluated in the destination model items separately. Moreover, the original operation can be replaced by this core operation. However, if the result of the core operation can vary in the destination model items, then it is possible that some of the destination model items produce false results. False result means in this context any result that does not equal with the result of the original operation. This means that a set operation, such as select cannot be used because it does not force the equality of the core operations in all of the model items. Nevertheless, the operation forall is true if and only if the core operation is true in all of the model items, thus, the result cannot vary in this case. Similarly, not exist is true, if the iteration condition is false for each destination model item. Note that the main difference between the previous (erroneous) subcases and this one is that the relocated constraint fails exactly only, when the original constraint fails. The relocated constraint is checked for each node of the new context, thus, the constraints are functionally equivalent. The statement of the proposition has proved.

If the multiplicity is the combination of Cases A to E, then the combination of the previous propositions must be used. The correctness of this case is proven by formalizing the relocation algorithm.

### 5.3.1.2 Formalizing the Relocation

This section introduces how OCLASM can be used in formalizing the constraint relocation driven by the aforementioned limitations. The algorithm is defined as a rule, which runs before eval is called with the outermost position of the invariant. The algorithm modifies the constraints by updating the GetPhrase and Child functions. To simplify the rule, it is worth defining formulas and helper rules:

\[
\varphi_{A, \text{Dest}}(\text{Phrase}, ID_A, \text{Dest}) : \text{Value}(\text{eval}(\text{Child(Phrase, 0)})) = ID_A \land \text{Value}(\text{Child(Phrase, 1)}) = \text{Dest} \tag{5.1}
\]

\[
\varphi_{A, x}(\text{Phrase}, ID_A, \text{Dest}) : \text{Value}(\text{eval}(\text{Child(Phrase, 0)})) = ID_A \land \text{Value}(\text{Child(Phrase, 1)}) \neq \text{Dest} \tag{5.2}
\]

If the original context is \(A\) and the new context is \(B\), then \(\varphi_{A, \text{Dest}}\) expresses that the Phrase is a navigation from \(A\) to \(B\), while \(\varphi_{A, x}\) expresses that the Phrase is a navigation from \(A\) to anywhere, but not to \(B\). The formulas use the information that the first child of a navigation (to model nodes and attributes) is the origin, while the second child is the name of the destination.

\[
\text{ForallCheck()} = \begin{cases} 
  \text{true, if } \exists \text{ModelPhrase} : \text{GetPhrase(ModelPhrase)} \neq \text{undef} \text{ and } (\varphi_{A, \text{Dest}}(\text{ModelPhrase}, A, B) \text{ or } \varphi_{A, x}(\text{ModelPhrase}, A, B) \text{ and } \text{Parent(ModelPhrase).PhraseType} \neq \text{"ForAll"}) \\
  \text{undef, otherwise}
\end{cases} \tag{5.3}
\]

This derived helper function returns true, if (i) there is navigation/attribute call from \(A\) to \(x\) (\(x \neq B\)), or (ii) there is a navigation from \(A\) to \(B\) and the outermost phrase in the constraint has a specific type ('ForAll').

\[
\text{GetSrc(ID, Dest)} = \begin{cases} 
  \text{Src, if } \exists ID_2 : ID_2 \in \text{To(ID, Dest)} \text{ and ID} \in \text{To(ID}_2, \text{Src)} \\
  \text{undef, otherwise}
\end{cases} \tag{5.4}
\]
This derived helper function obtains the name of the source side of a navigation. For example there is a navigation between $A$ and $B$, and we can navigate by using the destination name 'B'. This function obtains the reverse direction, namely, to source name, which is used in navigation from $B$ to $A$.

$$Top(Phrase) = \begin{cases} 
\text{Phrase, if } \text{Parent}(Phrase) = \text{undef} \\
\text{Top(Parent(Phrase)), otherwise}
\end{cases} \quad (5.5)$$

This derived function tries to find the outermost *Phrase* of the constraint by using a recursive method.

The helper rules shown below are used to automate the repeatable parts of the main relocation rule. The first rule (*AddChild*) creates a new *Phrase* with the given type and appends it to children of the parameter *Phrase*. This rule is used when the algorithm inserts new expressions into the constraint. The rule *AddBackNavigation* inserts a complete set of *Phrases* as a child of the parameter *Phrase*. The rule has two different modes according to the second parameter, the result of the *ForallCheck* function and the multiplicity on the source side. The first mode creates *Phrases* that call a variable with the name 'OrigSelf', while the second mode inserts *Phrases* implementing a navigation call from the new context to the original context.

**Algorithm 5.2** The *AddChild* rule

1: rule AddChild (Type, Parent, Idx)  
2: CHILD(Parent, Idx):= new(Pos)  
3: GetPhrase(CHILD(Parent, Idx)):= new(Phrase)  
4: GetPhrase(CHILD(Parent, Idx)).PhraseType:= Type

**Algorithm 5.3** The *AddBackNavigation* rule

1: rule AddBackNavigation(Phrase, WithForallCheck, O, D)  
2: if WithForallCheck and (FORALLCHECK() and Mul(O,D)[1]>1) then  
3: AddChild(VariableCall, Phrase, 0)  
4: AddChild(OrigSelf, CHILD(Phrase, 0), 0)  
5: else  
6: AddChild(NavigationCall, Phrase, 0)  
7: AddChild(SelfReference, CHILD(Phrase, 0), 0)  
8: AddChild(GetSrc(O,D), CHILD(Phrase, 0), 1)

The main algorithm (the rule *RelocateConstraint*) is shown in Alg. 5.4 (see next page). The *RelocateConstraint* operates as follows (sides of relation used in relocation are denoted by *Dest* and *Src*): (i) The rule forbids relocation when zero multiplicity is allowed on *Dest*, while (ii) it adds an encapsulating *IsEmpty* check when zero multiplicity is allowed on *Src*. (iii) If the multiplicity on *Src* allows more than one model item and the constraint needs to navigate back to the original context (FORALLCHECK), then the rule uses an encapsulating *forall* expression to reach all of the nodes of the original context. (iv) Inside the main loop each model query is examined. If multiplicity on *Dest* is one, then queries of the form $A.B$ are transformed to
B (ModelPhrase=Child(ModelPhrase,0)), while queries of the form A.x are transformed to B.A.x (via AddBackNavigation). (v) If multiplicity allows more than one, then A.B->Op(x) expressions are transformed into (x), when Op=forall, or into B.A.Op(x) if Op is not forall.

**Algorithm 5.4** The RelocateConstraint rule

1:  rule RelocateConstraint(O, D)
2:  if Mul(O,D)[2]=0 then
3:    RelocationError
4:  if Mul(O,D)[0]=0 then
5:    AddChild('IfExpr', undefined ,0)
6:    AddChild('IsEmpty', Child(undefined,0) ,0)
7:    AddBackNavigation(Child(Child(undefined,0), 0), false,O,D)
8:    Child(Child(undefined,0), 1):= Top(ModelPhrase)
9:    Child(Child(undefined,0), 2):= True
10:  if Mul(O,D)[1]>1 and ForAllCheck() then
11:    AddChild('ForAll', undefined ,0)
12:    AddBackNavigation(Child(undefined,0), false,O,D)
13:    AddChild('IteratorVariable', undefined ,1)
14:    Value(Child(undef,1)):'OrigSelf'
15:    Child(Child(undef,0), 2):=Top(ModelPhrase)
16:  for all ModelPhrase in {Phrase: GetPhrase(Phrase)<> undefined ∧ (Phrase.PhraseType= AttributeCall ∨ Phrase.PhraseType= NavigationCall)} do
17:    if ϕA.Dest(ModelPhrase,0,D) then
18:      if Mul(O,D)[3]>1 then
19:        if Parent(ModelPhrase).PhraseType =ForAll then
20:          GetPhrase(Parent(ModelPhrase)):= GetPhrase(Child(Parent(ModelPhrase),2))
21:        else
22:          AddBackNavigation(ModelPhrase, true,O,D)
23:        else
24:          ModelPhrase:= Child(ModelPhrase, 0)
25:      else
26:        AddBackNavigation(ModelPhrase, true,O,D)

**Proposition 5.19.** Constraint relocation based on Algorithm 5.4 always results in a correct relocation.

*Proof.* Appendix A. □

5.3.1.3 The Relocation Algorithm

The relocation algorithm (Algorithm 5.5) consists of two major parts: (i) searching for the optimal node (and RelocationPath) (Algorithm 5.6) and (ii) relocating the constraint if necessary. Relocation is based on the Algorithm 5.4, where the rule is invoked for each step of the RelocationPath separately.
Chapter 5. Optimized Metamodel-Based Constraint Validation

Algorithm 5.5 The RelocateConstraint algorithm
1: RelocateConstraint(Constraint, OriginalContext)
2: OptimalPath = SearchOptimalNode(Constraint, OriginalContext, NULL)
3: if OptimalPath ≠ OriginalContext then
4:   Update(Constraint, OriginalContext, OptimalPath)

The first part of the RelocateConstraint algorithm is based on the SearchOptimalNode function. This function checks the relocation requirements while searching (StepIsValid), thus invalid RelocationPath candidates are dropped as soon as possible. SearchOptimalNode uses a recursive breadth-first-search strategy to find every possible candidates. The external function CalculateSteps calculates the number of model queries in the case when the new context is located in N. Note that the algorithm StepIsValid is based on the algorithm presented and examined earlier.

Algorithm 5.6 The SearchOptimalNode algorithm
1: SearchOptimalNode(Constraint C, Node N, PathP)
2: minSteps = CalculateSteps(C, N)
3: optimumCandidate = Append(P, N)
4: for all CN in Connections(N) do
5:   if StepIsValid(C, CN) then
6:     LocalOptimum = SearchOptimalNode(Update(C, N, CN), Append(P, CN))
7:     LocalSteps = CalculateSteps(C, LocalOptimum.LastElement)
8:     if LocalSteps < minSteps then
9:       minSteps = LocalSteps
10:      optimumCandidate = LocalOptimum
11: return optimumCandidate

The result of SearchOptimalNode is the RelocationPath. If the new context and the old context are not the same, then the constraint is relocated and updated. As mentioned earlier, the relocation is based on path steps, thus, the algorithm updates the context declaration step-by-step.

5.3.2 Constraint Caching

5.3.2.1 The Algorithm

Using the aforementioned algorithms, the number of model queries is still not minimal. This section presents an algorithm that focuses on reducing the number of redundant model queries instead of changing the constraint expression.

In compiler optimization, an occurrence of the expression $E$ is called a common subexpression if the value of $E$ has previously been computed, and it has not changed since then [Aho et al., 1988]. In these cases, recomputing the expression can be avoided, because the value of the expression is already known.

Proposition 5.20. In OCL constraints, navigation steps and attribute references are always common subexpressions during a constraint evaluation.
Proof. OCL specification defines the constraints as restrictions on one or more values, but these restrictions cannot have any side-effects. This means that the constraint cannot change the model and – as mentioned earlier – we assume that the model does not change under the evaluation. Thus, the computed values, the results of model queries can always be reused.

The presented idea is the basis of the second optimization algorithm. On the one hand, caching the model items can eliminate the redundant model queries in the constraint expressions. On the other hand, the more attribute or navigation is cached, the more memory the cache requires. Thus, only those expressions are cached that are referenced more than once. The optimization algorithm (the ReferenceCaching algorithm) has two main steps: (i) obtaining statistical information about the model references (GetCommonReferences algorithm), and (ii) caching the evaluation expressions (CachingManagement algorithm).

Collecting the statistical information set from the whole constraint expression is not straightforward, because sometimes only partial validation is required on a model. Thus, the caching algorithms are used at the context level, constraints with common context are analyzed and cached together, while the statistical information of the different contexts are separated.

The GetCommonReferences algorithm is shown in Alg. 5.7. It applies an offline control flow analysis on the constraint. The algorithm uses a breadth-first search to traverse the syntax tree recursively. It processes the attributes, the navigations and the control flow expressions.

**Algorithm 5.7 The GetCommonReferences algorithm**

```plaintext
1: GetCommonReferences(CurrentExpression)
2: if ExpressionType(CurrentExpression) is AttributeDefinition then
3:   IncreaseReferencePath(CurrentExpression)
4:   return
5: if ExpressionType(CurrentExpression) is NavigationStep then
6:   IncreaseReferencePath(CurrentExpression)
7:   for all CurrentExpression.Children as navStep do
8:     GetCommonReferences(navStep)
9:   return
10: if ExpressionType(CurrentExpression) is ControlFlowExpression then
11:   minReferences = GetMinimumReferencesForExecutionPath()
12:   for all minReferences as modelItem do
13:     IncreaseReferencePath(modelItem)
14:   return
15: for all CurrentExpression.Children as child do
16:   GetCommonReferences(child)
```

The attribute calls and navigation expressions increment the statistics of their path reference (using the IncreaseReferencePath function). To minimize the number of queries, the algorithm increments not only the reference of the full path, but also the references of the path steps. For example, the expression `self.employee.wife.Name` increases the statistics of four entries: `self`, `self.employee`, `self.employee.wife` and `self.employee.wife.Name`. The statistics contains even the `self` element, because it is not cached if it is referenced only once. In
the algorithm this is why the child expressions, namely, the steps of the path are recursively checked in the case of NavigationSteps. Increasing the reference counter of the path steps is useful if two expressions have a common subset in the navigation steps, for example, in the expression \( \text{self.employee.wife.Name} = \text{\ Respons} + \text{self.employee.Name} \), the path self.employee is used twice.

The control flow expressions are complex expressions that have several execution paths, for example conditional expressions, or loops. These expressions can affect the number of the references according to their execution parameters. The problem is that these execution parameters are usually obtained at run-time only. Therefore, the algorithm obtains the minimum number of the references for each referenced objects for each execution paths. For example, in case of the conditional expressions this means that both branches are processed, statistical information is collected for both branches, and then the results are compared. For each model reference path (attribute, or navigation reference), the minimum number of references is obtained and placed into the global statistical information set.

As the result of GetCommonReferences algorithm, the compiler has reliable statistical information based only on an offline algorithm. CachingManagement algorithm uses this information to handle caching. CachingManagement algorithm differs from the previously presented algorithms, because it affects the generated source code directly instead of affecting the syntax tree only. Each time the compiler generates a navigation step or an attribute query, the statistics are checked, and a cache (a local variable) is created if required. This variable obtains the appropriate value from the model if it has not been read before, or returns the value from the cache if it is not the first query. If the model reference is not cached, the code generator will create a conventional source code for it.

Proposition 5.21. If the ReferenceCaching algorithm is used, the number of the applied queries is less than or equal to that without the algorithm. Additionally, each attribute or navigation cached by the algorithm reduce the number of the model queries, thus, no unnecessary caching is applied.

Proof. The GetCommonReferences references algorithm is applied at design-time, it does not raise the number of the queries during the evaluation. The CachingManagement algorithm handles two types of model references: the cached, and the uncached references. The source code and thus, the number of model queries of uncached model references is the same as in the unoptimized code. The cached references execute the appropriate model query only if the required value is not in the cache, i.e. it has not been read before. Therefore, neither the uncached nor cached references increase the number of the model queries.

The GetCommonReferences algorithm is executed for each referenced context. If the context contains an expression that has several possible execution paths, then every path is examined, and for each model attribute and navigation the smallest number of references is stored. The sequential execution paths are examined step-by-step, and the statistics is increased if required. As result, the statistics contains the minimum number of the references in the context for every model item (attribute, or navigation). The CachingManagement algorithm creates caching code only for the model references that have greater statistical index than one. Since the statistics
contains the minimum number of the references of the current item, no unnecessary caching is performed.

To have a precise comparison of constraint evaluation time without and with optimization, we have measured the time required by applying a single DB-based model query and the time required by a cached-based query. Referencing a cached value required approximately five magnitude less time regardless of iteration counts (the database could not cache the query efficiently).

5.3.2.2 Formalizing the Algorithm

In the followings, we prove that the application of the caching algorithm results always in correct constraint validation. To achieve this, we formalize the CachingManagement algorithm in OCLASM. It is not necessary to formalize the GetCommonReferences algorithm, because it is a compile-time algorithm that does not change the model, the constraint or the evaluation of the constraint.

CachingManagement algorithm is formalized by creating an embedding function for each monitored function of OCLASM (IsModelItem, AttrValue, Meta, To, Mul). These embedding functions have the same name and same input parameters. They store the result of previous queries locally and apply the model query only if this local result is not set (see the example below). In order to distinguish the embedding and local cache functions from the original, we use labeling (e.g. ToEmbedding).

\[
\text{IsModelItem}^{\text{Embedding}}(ID) : \\
\text{if } (;\text{IsModelItem}_{\text{Cache}}(ID) = \text{undef}) \; \text{IsModelItem}_{\text{Cache}}(ID) = \text{IsModelItem}_{\text{Original}}(ID) \\
\text{return } \text{IsModelItem}_{\text{Cache}}(ID)
\]

When evaluating the constraint, these new, embedding functions are used instead of the original monitored functions. The vocabulary of OCLASM is extended by the following derived functions: \text{IsModelItem}^{\text{Embedding}}, \text{AttrValue}^{\text{Embedding}}, \text{Meta}^{\text{Embedding}}, \text{To}^{\text{Embedding}}, \text{Mul}^{\text{Embedding}}, and by the following dynamic functions: \text{IsModelItem}_{\text{Cache}}, \text{AttrValue}_{\text{Cache}}, \text{Meta}_{\text{Cache}}, \text{To}_{\text{Cache}}, \text{Mul}_{\text{Cache}}. The initial value of dynamic cache functions is \text{undef}, which is set by the CheckModelInvariants rule (Section 5.2.2) each time an invariant is checked against a new model item. We refer to the extended formalism as \text{OCLASM}_{\text{Cache}}.

**Proposition 5.22.** The constraint evaluation based on the CachingManagement algorithm is always correct.

**Proof.** Given \( F \), which is one of the monitored functions of OCLASM. If \( F^{\text{Embedding}} \) is requested during the evaluation of the constraint and \( F_{\text{Cache}} = \text{undef} \), then the result is directly retrieved from the underlying model by \( F_{\text{Original}} \). This means that \( F^{\text{Embedding}} \) always equals \( F_{\text{Original}} \), if \( F_{\text{Cache}} \) is unset.

If \( F_{\text{Cache}} \) is set, then it contains the result of a previous model query. However, the model does not change during evaluation. We consider by definition that the model is not changed by external functions. Moreover, OCLASM use monitored functions only to retrieve the data
from the underlying model. We do not allow changing the value of $F_{Cache}$ outside $F_{Embedding}$ to anything, but $undef$ by the definition of CachingManagement algorithm. This means that $F_{Cache} = F_{Original}$ is always true, the statement of the proposition is proven.

5.4 Chapter Summary

The ability to define precise, complete and unambiguous visual languages is a natural need. The lack of preciseness stemming from the visual definition of these languages can be solved by specifying textual constraints and attaching them to the model items. A formal language is needed to describe these constraints and ensure preciseness. OCL is a formal constraint language, possibly the most popular of all others. OCL was originally invented to extend UML models only, but nowadays, it is used frequently in other domains as well.

In this chapter, the concepts of a metamodel-based optimizing constraint validation approach have been presented. Firstly, a dialect of OCL has been introduced that supports the attribute structures of n-layer metamodeling. Secondly, a new formalism – OCLASM – has been given for OCL. The formalism was created to solve the lacks of existing OCL formalisms [Flake, 2004] to describe OCL dialects in a simple way and to model the dynamic behavior of constraint evaluation. The new formalism is not based on set theory, as the original formalism was, but it uses the popular ASM formalism [Böerger and Stärk, 2003]. By using the decomposition and arbitrary abstraction level of ASMs, the new formalism is much more compact, while it is as precisely defined as the old one. OCLASM was built to support the OCL dialect mentioned earlier.

Since the constraint validation often lies at the heart of modeling, it is essential to have an efficient constraint evaluation method. The chapter proposed two algorithms that can optimize OCL constraints.

The first algorithm (RelocateConstraint) tries to find the optimal context for each constraint and relocate the constraint to it. However, multiplicities can strongly limit the relocation, thus, it is discussed in which multiplicity combination and exactly how the relocation can take place. The RelocateConstraint algorithm is formalized using OCLASM, and its correctness is proven formally.

The second algorithm (ReferenceCaching) is based on the fact that OCL constraints cannot change the underlying model. It realizes a constraint caching method with offline control flow analysis-based reference counting to avoid unnecessary cached attributes. The correctness of the algorithm has also been proven formally.

The constraint optimization algorithms provided in the chapter definitely improve the performance of constraint evaluation and therefore they improve the modeling as well. The algorithms are useful not only in metamodel-based OCL dialects, but in any other domain, where the constraint evaluation is of great importance.
Chapter 6

Truly Parallel Graph Transformation

The efficiency of model-based development strongly depends on the efficiency of model processing techniques. Graph rewriting-based model transformation is a popular and intuitive way of processing graphical models. Although the visual representation of transformations is beneficial, the performance of these methods does not really scale up in case of large input models.

More precisely, in the general case, the complexity of a single rewriting step is $O(n^k)$, where $n$ is the size of the input graph, and $k$ is the size of the pattern defined in the rewriting rule. There exist several approaches which can reduce this complexity in special cases, but these solutions are not applicable in general. The performance issue of graph transformations can be critical in complex real-life examples.

Reducing the time required by the transformations can be reached not only by accelerating the pattern matching, but also by applying the steps of the transformations in parallel. Naturally, this acceleration is not always possible, since transformation steps have the tendency to depend on each other.

This chapter presents the theoretical and practical basis of a truly parallel model transformation engine. Section 6.1 elaborates the background, namely the theoretical constructions used in the approach and the related work. Section 6.2.2 presents an efficient solution to find transformation steps, which can be applied in parallel (transformation-level parallelism). Section 6.2.3 introduces a distributed algorithm to compute the match of a single rewriting rule in parallel (rule-level parallelism). Section 6.2.4 elaborates a unified parallel transformation approach that benefits from the advantages on both the transformation-level and rule-level parallel algorithms. Section 6.3 summarizes the results presented in the chapter and concludes.

6.1 Backgrounds and Related Work

The theoretical background of algebraic graph transformations has been presented in Section 3.3. This section gives an overview on the approaches aiming to accelerate matching, and on some of the most remarkable tools supporting model transformation techniques.
6.1.1 Acceleration of Graph Rewriting

Graphs and graph related algorithms were always a central issue in mathematical research. There exist numerous approaches that aim to accelerate certain graph algorithms by using parallelism. Parallelization has been successfully used in several classical problems of graph theory, such as searching for Minimum Spanning Tree [Bentley, 1979], or depth-first search in directed acyclic graphs [Ghosh and Bhattacharjee, 1984]. However, there does not exist such a well-accepted solution for parallelizing pattern matching (i.e. for subgraph isomorphism).

The paper [Fatta and Berthold, 2005] is using parallel algorithms in analyzing molecular compounds. The author recognizes and classifies problems in connection with pattern matching. Although the paper is not about graph rewriting, the basis of the presented problems is the same: it is not possible to give an appropriate approximation for the minimum or maximum bounds for the running time of subtasks. The workload and the time required by the clients can vary in a wide spectrum. In order to achieve optimal load balancing, the approach uses a quasi-random polling. Similar solution is used in the second rewriting rule-level parallelism algorithm presented in this thesis.

The paper [Eppstein, 1994] presents a solution for subgraph isomorphism in linear time. The solution considers that the graph is planar, which is generally not true in case of model graphs. Another approach [Dodds and Plump, 2006] presents a constant time algorithm for subgraph isomorphism, but the approach achieves this result by having a concrete rule, not by describing the general case. Moreover, constant time means in this case that the maximum number of outgoing edges and the number of all edges in the graph can be replaced by a constant, since we can count them in a particular case.

6.1.2 Model Transformation Approaches

The book [Fuller et al., 2000] gives a precise overview about the existing parallel algorithms in graph theory. The description is not focusing on graph transformations, instead, it introduces the commonly used mathematical background in the field of graph algorithms.

PROGRES [Ehrig et al., 1999a] is a graph transformation tool supporting incremental matching technique by attribute updates [Bunke et al., 1990]. PROGRES creates an evaluation order of the pattern variables based on their place in a dependency graph. This dependency graph is modified during the matching to follow the changes of the models (e.g. deleting a model node).

Fujaba [Fischer et al., 1998] is an open source UML CASE tool that provides a rule-based visual programming language for manipulating the object structure, based on the paradigm of graph transformations. Fujaba uses a simplified UML collaboration diagram to specify graph transformation rules. Graph rewriting rules are based on local search, which starts from matching a single node and extends the matching to the neighboring nodes and edges. Fujaba uses incremental matching [Varró et al., 2006] to accelerate the transformations by storing every match explicitly in a matching tree.

Although PROGRES and Fujaba do not use parallel matching algorithms, the construction method of the search plan can be used as a heuristic algorithm in our parallel approach. The main problem with incremental matching is that it is efficient only if a small part of the graphs
are modified during the incremental execution. However, this is just a small subset of graph transformations.

In VIATRA2 (VIsual Automated model TRAnsformations) [Varró and Pataricza, 2003], the instantiation is based on mathematical formalisms, called Visual Precise Metamodeling. The transformation language of VIATRA2 supports type checking, negative patterns, attribute conditions and traditional pattern matching issues. The model and rule constraints can be expressed by graph patterns with arbitrary levels of negation. VIATRA2 uses abstract state machines (ASM) to define the control flow of the system. VIATRA2 can rewrite multiple matches of a rewriting rule in parallel, but it does not support distribution on multiple computers, or any kind of transformation-level parallelism.

Distributed Graph Transformation System (DGTS) [Ranger and Lüstraeten, 2006] introduces a new approach by creating a visual language to handle distributed transformation. On the one hand, this solution grants a high level of efficiency, since the specification can exactly define how the distribution should be applied. On the other hand, this also means that the sequential and parallel transformations are distinguished at the specification level and at the implementation level as well. In contrast, our approach can find the parallelization possibilities in the transformation automatically at run-time, without requiring to prepare or modify the transformation.

The paper [de Lara Jaramillo et al., 2004] introduces an interesting solution for parallel model transformations based on amalgamated rules (Section 3.3.4). The main idea is that rules of the transformation, which are not parallel independent can be executed concurrently by creating a new rule that contains the common parts of the rules and then adding actions that describe the changes, which are not common. Using this idea, the original transformation is converted to iteration schemes. Although the approach presents a straightforward solution, it is hard to follow in metamodel-based rewriting, where multiplicities can also be used to define the LHS and RHS of the rule. In this case, it is hard to find the common subrule, since a model item of the rule can refer to more than one model item in the host model. Moreover, the approach does not support rule-level parallelization.

Although there exist many model transformation approaches as introduced in this section, there is a need to enhance the performance especially to support large models. Reducing the original complexity is possible only in certain cases, thus a new, universal solution is required. Since computer clusters, local computer networks are often used in complex calculations, it is an obvious idea to apply the transformations in parallel by using a distributed environment. However, there does not exist such a truly parallel solution at the moment, which can explore parallelization possibilities automatically by dynamically adopting to the number of available workers.

6.2 Parallel Model Transformations

This section presents transformation-level and rule-level parallelism approaches and a combined model transformation engine supporting both levels of parallelism. All three approaches are based on a server-client architecture, where the server coordinates and synchronizes the clients that are applying the computation in parallel. In this section, we do not define the exact
implementation of the architecture though, in order to create a flexible solution that is universally applicable. For more details on our concrete implementation please refer to Section 7.

6.2.1 Basic Definitions

Our model transformation approach uses metamodel-based rewriting rules. Recall from Section 3.3 that this means that instead of a subgraph isomorphic to LHS, an instantiation of LHS must be found in the input graph. To distinguish between the rule definition and its instantiation, we refer to them as metamodel-based rules and instance model-based rules respectively.

In order to describe metamodel-based rewriting, we define helper functions applicable on model graphs. Model graphs are labeled, directed graphs as defined in Section 3.3. We define a special label Stereotype for each edge. We use this label to set the type of relation (e.g. 'Inheritance', or 'Containment') represented by the edge. Moreover, we define a typeof function between the items as \( \text{typeof} : G^V \to G^M_X \), where \( X \in \{V, E\} \), \( G \) represents the model, while \( G^M \) represents the metamodel. This function obtains the meta item of a model item.

Using the Stereotype label and the typeof function, we define four functions. The first function \( \mathcal{S}(G) \) obtains all nodes and edges contained by the graph \( G \). The function \( \mathcal{I}(X) \) maps the inherited types of the base type \( X \), the result of the function is a set of objects (nodes or edges) including \( X \). The third function \( \mathcal{P}(X) \) applies the reverse direction, it results in a set of model items from which \( X \) inherits. The fourth function \( \mathcal{T}(X) \) obtains the type (the metamodel item) of a model item (a node or an edge), the result is always exactly one model item.

\[
\begin{align*}
\mathcal{S}(G) &= G^V \cup E^V \\
\mathcal{I}(X) &= \{ Y : \exists e : s(e) = Y \land t(e) = X \land \text{Stereotype}(e) = '\text{Inheritance}' \} \\
\mathcal{P}(X) &= \{ Y : \exists e : s(e) = X \land t(e) = Y \land \text{Stereotype}(e) = '\text{Inheritance}' \} \\
\mathcal{T}(X) &= \{ P : \text{typeof}(X) = P \}
\end{align*}
\]

Note that the instantiation preserves the source and target functions by definition w.r.t. to the \( \mathcal{I} \) function, thus

\[ \forall e : \mathcal{T}(s_D(e)) \in \mathcal{I}(s_D(\mathcal{T}(e))) \]  

and

\[ \forall e : \mathcal{T}(t_D(e)) \in \mathcal{I}(t_D(\mathcal{T}(e))). \]

By the definition of metamodel-based rewriting it is always true in metamodel-based rewriting rule \( R \) that for each \( R_X \) node and edge of the rule and for the corresponding \( X \) node, or edge of the concrete (instantiation level) match \( m(X) \) the formula

\[ \mathcal{T}(m(X)) \in \mathcal{I}(R_X) \]

is satisfied. The intuitive meaning of this formula is that matching preserves the \( T \) function w.r.t. the inheritance.
In order to simplify the notation of metamodel-based rewriting rules, we will denote them with an \( M \) index. For example, \( R^M \) is a rule built from metamodel terms, while \( R \) denotes its instantiation.

### 6.2.2 Transformation-Level Parallelism

The complexity of a single rewriting step is \( O(n^k) \), where \( n \) is the number of model items in the input model, while \( k \) is the size of the LHS. This means that the complexity of the whole transformation is

\[
O\left(\sum_{i=1}^{r} n_i^k\right),
\]

where \( r \) is the number of transformation steps. By applying the transformation steps in parallel, this complexity can be reduced, but parallelism possibilities are limited by the dependencies between the steps. Moreover, the strict order of rewriting rules are determined by the transformation control flow in our approach (as described in Section 3.3.6).

#### 6.2.2.1 Dependency Between the Rules

A natural idea would be to limit the scope of rules executed in parallel only to those rules which are independent. However, this solution is hard to realize. The rules are defined by metamodel elements, thus, having the definition of the rules is not always enough to decide whether they conflict. Here conflict is defined as follows:

**Definition 6.1.** An instance model-based production rule \( r_k \) of a strictly ordered transformation \( t \) is in conflict (or simply conflicts) with another instance model-based rule \( r_i \) of \( t \) if there is a rule sequence leading from \( r_i \) to \( r_k \) in \( t \), and \( r_i \) is not sequentially independent from \( r_k \).

By examining two rules that have already been matched in the host model, we can decide whether they conflict, whether they are executable in parallel. In this way, we can always avoid incorrect execution of the rules. Nevertheless, this examination method is not efficient, since it must be applied for each possible rewriting rule pairs of the transformation. Furthermore, if there is a rule sequence consisting of five rules \( (r_1 \rightarrow \ldots \rightarrow r_5) \), then applying the rule \( r_5 \) must be suspended until all the preceding rules have their match in order to check whether \( r_5 \) has a conflict with them.

Parallel execution of metamodel-based rules may lead to incorrect transformations. The extension of Parallelism theorem for metamodel-based rules ensures that parallel and sequential parallelism equal in case of metamodel-based rewriting rules. This means that the following two statements are identical for all possible rule pairs of a rule sequence: (i) the rules can be swapped, (ii) the rules can be executed in parallel. The definition of rule conflicts can be generalized to support metamodel-based rewriting rules as follows:

**Definition 6.2.** In a metamodel-based transformation, we say that two rewriting rules are in metaconflict, if one of the rules deletes, or creates a metamodel element \( X \), for which \( I(X) \cap S(L_2) \) is not empty, where \( L_2 \) is the LHS definition of the another rule.
Proposition 6.3. If two subsequent metamodel-based rewriting rule in a rule sequence is applied in reversed order and the rules are not in metaconflict, then the result of the rule sequence does not change.

Proof. Appendix A.

According to the definition, the examination of the metaconflict relation can be applied offline, it requires the transformation definition only, not the host model. Rules, for which the metaconflict relation does not apply and which are direct neighbors in the transformation can always be executed in parallel. Note that this statement is not true in general for two arbitrary chosen rules. For example, there is a transformation $t : r_1 \to r_2 \to r_3$, in which $r_1$ is in metaconflict with $r_2$, $r_3$ is in metaconflict with $r_2$, but the relation does not apply on the $r_1, r_3$ pair. Applying $r_1$ and $r_3$ in parallel and applying $r_2$ only thereafter can result in an incorrect execution of the transformation because of the metaconflict between $r_2$ and $r_3$.

Rules are applied only if all possible preceding rules have finished, which are in metaconflict with the current rule. This method ensures correct results for parallel execution of transformations. Using this approach, we can build independence blocks from the rules. Independent in this context refers to the fact that rules inside a block are applicable in parallel. In order to achieve this, blocks contain a set of rules, which were adjacent rules in the original transformation and which are not in metaconflict with each other. Before applying the transformation, it is converted to a sequence of independence blocks. Note that this conversion can be applied offline. During the conversion, construction of a block is finished and a new block is started, when the next rule would be in metaconflict with a rule of the block.

By converting the original transformation to a sequence of independence blocks, we can adopt to the capabilities of parallel execution. Each rule in a block is sequential and therefore parallel independent from any other rules of the block, thus, the rules of a block can be applied in parallel.

Proposition 6.4. Assume an independence block with more than one rule $r_1, r_2 \ldots r_k$. Let $t_x$ denote the cost (time) the application (matching and rewriting) of the rule $r_x$. We can select the rule $r_{\text{max}}$, which has the largest cost for $\forall j \neq i : t_j \leq t_{\text{max}}$, and a second rule $r_{\text{max}2}$, which requires the largest cost not counting $r_{\text{max}}$.

The cost of applying an independence block can be expressed as

$$O(t_{\text{max}} \times (1 + \lceil \frac{k-1}{c} \rceil) + h \times k),$$

where $k$ is the number of rules in the block, $c$ is the number of clients and $h$ is an upper bound for the network cost on applying a rule.

Proof. We show that the proposition is correct by calculating the cost of the worst case and showing that it is exactly the expression of the proposition. The largest cost arises if $r_{\text{max}}$ is the last rule, for which we start the calculation. Otherwise, if $r_i \neq r_{\text{max}}$ is the last rule, then the calculations need less time ($t_i \leq r_{\text{max}}$).

However, this means that we have $k - 1$ rules to compute, the cost of the rules is at most $t_{\text{max}2}$. Since we have $c$ clients and all rules are independent inside the block, thus, the cost of
application the block is
\[ O\left(\left\lceil \frac{k-1}{c} \right\rceil * t_{\text{max}}\right). \] (6.10)

However, \( t_{\text{max}} > t_{\text{max}2} \), thus the calculation of all rules is in the order of
\[ O\left(\left\lceil \frac{k-1}{c} \right\rceil * t_{\text{max}} + t_{\text{max}}\right). \] (6.11)

Then, the cost of network traffic must also be added. This results the expression of the proposition, thus, the approximation of the cost is correct.

As the proposition shows, it is important to have rules that have approximately equal cost. If this condition is not satisfied, parallel execution is not necessary efficient. However, this issue is solved by the rule-level parallelism presented later.

### 6.2.2.2 Handling Possibly Conflicting Rules

Although starting a new block only after the preceding block has been finished results always in a correct parallel execution, the number of tasks applicable in parallel are strongly limited by the size of the blocks. Using block construction rules based on the metaconflict relation is an all-or-nothing strategy, which is not efficient in most of the cases. Instead, it is worth to overlap the execution of different independence blocks. Obviously, this means that it may be possible to construct a host graph, where the rules applied in parallel are parallel dependent, they may conflict. Therefore, there are two issues to solve: (i) we have to avoid parallel execution of likely conflicting rules and (ii) we have to resolve the conflicts, when they occur after all. In order to avoid conflicts, our approach uses heuristic algorithms.

#### Heuristic Algorithms

Since different transformation characteristics need different heuristic algorithms to be optimal, thus, heuristics are not hard coded in our approach, but they are realized in external components. These components can be added to the transformation engine dynamically, moreover, the list of components can be extended as well. When executing a transformation in parallel, the user, or the transformation engine selects a heuristic configuration to use. This configuration is a weighted list of the heuristic algorithms, where the weights represent how important the given heuristic algorithm is in comparison with others. When overlapping is applied between two independence blocks, then all heuristic algorithms are queried to produce a preference lists. These preference lists are summarized w.r.t. the weight of the algorithms. The algorithm used in our approach is presented in Alg. 6.1.

Firstly, the algorithm sets the index of all unfinished rules in the preceding independence rule to 0. Index is a numerical value in this case, which is used to order the rules (to choose the rule to be executed in overlapping style). Secondly, the step in Line 3 calls each heuristic algorithm and obtains their preference list. Finally, the minimum value of the global preference list is retrieved (Line 5-7).
Algorithm 6.1 The \texttt{GetNextRule} algorithm

\begin{algorithm}
\begin{algorithmic}[1]
\Function{GetNextRule} \EndFunction
\ForAll{\texttt{Rule} \IN NextBlock} \EndFor
\If{Unfinished(\texttt{Rule})} \EndIf
\ForAll{\texttt{Algorithm} \IN HeuristicConfiguration} \EndFor
\texttt{PrefList} = \texttt{GetPreferenceList}(\texttt{Algorithm}) \EndFor
\ForAll{\texttt{(Rule}, \texttt{Index}) \IN PrefList} \EndFor
\GlobalList[\texttt{Rule}] += \texttt{Index} \EndFor
\MinValue = 0 \EndFor
\ForAll{\texttt{Rule} \IN NextBlock} \EndFor
\If{Unfinished(\texttt{Rule}) \AND \MinValue < \GlobalList[\texttt{Rule}]} \EndIf
\MinValue = \GlobalList[\texttt{Rule}] \EndFor
\SelectedRule = \texttt{Rule} \EndFor
\end{algorithmic}
\end{algorithm}

We have created several heuristics. To simplify the description of the algorithm, we use the expression \textit{old} block to refer to the current block under execution and we refer to the subsequent block as the \textit{new} block.

The first and simplest algorithm (\textit{StrictHeuristic} algorithm) disables any kind of overlapping. The algorithm is used if metaconflicts likely mean a real conflict.

Heuristic algorithms are often calculated based on the number of metaconflicts between the new, overlapping rule and the rules of the old independence block. This number is referred to as \#\textit{MC}. The second algorithm (\textit{MCMinimizer} algorithm) minimizes \( k_{MC} \). More precisely, it selects the rule from the new block, for which \#\textit{MC} is minimal. The algorithm is very simple, it calculates \#\textit{MC} for each rule of the new block and selects the optimal one.

However, this second algorithm can be optimized by calculating the number of metaconflicts only to unfinished rules of the old block (\#\textit{MCopt}) and not to all rules of the block. This modified algorithm is referred to as \textit{OptimizedMCMinimizer} algorithm. Since changes caused by the previously finished rules are surely applied, thus, conflict cannot occur in this case.

The optimization of \textit{OptimizedMCMinimizer} algorithm means that the execution order of the rules in the old block is also important. It is worth to execute those rules of the old block first, which are in metaconflict with several rules of the new block. The exact number of metaconflicts in this direction is referred to as \#\textit{MCRev}. By applying rules with higher \#\textit{MCRev} first, we can reduce the average \#\textit{MC} during the calculation and reduce the degree of dependency between the blocks.

To illustrate the difference between the approaches, Fig. 6.1 shows an example, where arrows denotes the metaconflicts between the rules. The \textit{MCMinimizer} algorithm is used to select a rule from the new block, when all rules of the old block is currently processed or previously finished. The algorithm selects \( r_5 \) in this case, because it has the smallest number of metaconflicts with rules of the old block. However, \#\textit{MCRev} is used when there are rules in the old block, which have not been started yet. In this case, our goal is to execute that rule first, which has the largest number of metaconflicts with the new block. According to this, firstly, we would compute \( r_2 \), then \( r_1 \) and finally \( r_3 \).
The *BidirectionalMCMinimizer* algorithm combines the two techniques. The pseudo code of the algorithm it is shown in Algorithm 6.2. In order to allow setting how important the forward and backward *metaconflicts* are, we can pass two parameters to the algorithm. Note that the algorithm is for generic use, it is not based on a special property of the transformation.

**Algorithm 6.2** The BidirectionalMCMinimizer algorithm

1: function GetNextRule(Weight\_Back, Weight\_Fwd)
2: MaxIndex = -1
3: PreferredRule = NULL
4: for all Rule In PreceedingBlock do
5: \[ \text{Index} = \#MC(Rule) \times \text{Weight\_Back} + \#MCR\text{ev}(Rule) \times \text{Weight\_Fwd} \]
6: if Index > MaxIndex then
7: \[ \text{MaxIndex} = \text{Index} \]
8: PreferredRule = Rule
9: return PreferredRule

Many other algorithms can be created to optimize the rule selection. For example, simple rules of the new block can be preferred in order to save the cost of rollbacks, when conflict occurs. However, this is not always optimal, since complex rules are given lower priority and it can be time consuming to calculate them after the old block has finished. Nevertheless, preferring complex rules of the new block can lead to large rollback cost. The algorithm that is optimal in a case can be inefficient in other cases. One of the most important advantages of our approach is that it allows creating heuristic algorithms for the transformation engine and configure them. This means that the approach supports a default way of optimized rule overlapping, but it can be overridden at run-time. The feature is especially useful in case of special domains, such as supporting refactoring in UML class diagrams.

**Conflict Handling**

Our approach uses heuristics to reduce the number of conflicts between the application of rules, however, conflicts may occur. Usually, applying a rewriting rule consists of two main steps: (i) searching a match in the host graph and (ii) modifying the host graph according to the definition of the rule. To avoid incorrect transformations, we allow applying only the first step
for overlapping rules. More precisely, rewriting is applied in the appropriate client, the modified
host graph is calculated, but the original graph is immediately restored as well. This method is
necessary because our approach does not support analyzing the Imperative OCL code describing
the changes (Section 3.3.6). Instead, the code is executed and changes are noticed.

As mentioned earlier, matches of overlapping rules are allowed. When a match – and the
dependent modification list – is submitted to the server, the match is set to suspended mode.
Each time a rule from the old block is finished, suspended rules are re-checked by the following
algorithm:

Algorithm 6.3 The ResumeCheck algorithm

1: function ResumeCheck(Rule)
2: for all \( R \) in SuspendedRules do
3: if \( \text{ReferencedMetaItems}(R) \cap \text{ReferencedMetaItems}(\text{Rule}) \neq \emptyset \) then
4: if not CheckMetaConflicts(\( R \)) then
5: ApplyChanges(\( R \))
6: ResumeCheck(\( R \))

After submitting the match and the modifications of a rewriting rule to the server, the client
does not wait for the result of the conflict handling algorithm. Instead, it requests immediately
a new rule to compute.

When the second phase of the execution of a rule is allowed, the modification list is submitted
to all clients. Clients receiving the list apply the changes locally. Currently, our approach uses
higher priority for modifications, than for matching. This means that the matching procedure
of a rule is terminated if a modification list is received. The modifications are applied and the
match is restarted. In order to avoid unnecessary terminated matches, clients check whether the
received modification list and the current rule have common meta elements. If not, applying the
modifications is suspended until the match is calculated.

6.2.3 Rewriting Rule-Level Parallelism

Theoretically, the complexity of rewriting rules can be described as \( O(n^k) \) in general, where \( n \)
is the number of model items in the host graph, while \( k \) is the number of model items in the LHS
pattern of the rule. However, in practice, two rules with the same number of model items often
require considerably different amount of time to find the match. The reason for this is based on
the fact that \( n^k \) steps (item check) are required in the worst case, but only 1 step in the best
case. Although the worst and the best cases occur rarely, the differences can be remarkable. In
the followings, the amount of maximum steps is refined, then two parallelization methods are
presented to reduce the differences between the matching time of rules.

6.2.3.1 Refining the Maximum Amount of Steps

There are several properties of rewriting in our approach, which can reduce the overall complex-
ity. Firstly, a node of the host graph cannot be matched to more, than one node in the LHS
(Section 3.3). This means that the we need maximum \( \binom{n}{k} \) steps instead of \( n^k \).
Secondly, the host graph representing the host model can be partitioned along the metatype of the model items. Note that the partitions can be constructed offline, when creating the rule. The partitions can overlap due to inheritance only. The LHS of the selected rule can be partitioned similarly. As result, for each \(rn\) node in the instantiation-based rule, we have a relation \(S(rn)\) defined as

\[
S(rn) : \{ \forall x \in S(rn) : x \in I(\mathbb{T}(rn)) \}.
\] (6.12)

The significance of (6.12) is that, usually, \(S(rn)\) contains much less element, than \(n\). The maximum steps can be expressed in this case as

\[
\prod_{i=1}^{k} (S(rn_i) - \sum_{j=1}^{i-1} N(rn_i, rn_j)),
\] (6.13)

where \(k\) is the number of nodes in the selected rule, and \(N\) is a function defined as:

\[
N(l, m) = \begin{cases} 
1, & \text{if } \mathbb{T}(m) \in I(\mathbb{T}(l)) \\
0, & \text{otherwise}
\end{cases}
\] (6.14)

Although further refinement is possible, (6.13) shows that the size of metamodel-based partitions can heavily affect the maximum number of steps. This information is the basis of the first rule-level parallelization algorithm.

6.2.3.2 Parallelization Based on Fix Pivot

Based on (6.13), it is useful to start matching with a rule node that has the least compatible nodes in the host graph. Recall that compatible host nodes are nodes whose metamodel is included in \(I(r)\), where \(r\) is the metamodel term of the selected rule node. We can collect a list of possible starting points for matching, which are the set of compatible nodes in the host graph. These (selected) starting nodes in the host graph are referred to as pivot points for the rule. If the task is to find all possible matches of a rule (not only the first) and starting from different pivot points cannot produce the same match, the task can be efficiently parallelized by giving different pivot points to the clients working in parallel.

**Proposition 6.5.** Assume an arbitrary, metamodel-based rewriting rule \(R\). Furthermore, given a rule node \(r\) of \(R\). The node \(r\) is chosen arbitrarily except that (i) there is at least one edge leading to \(r\) and (ii) all of the edges \(e_i\) leading to \(r\) allow only ExactlyOne multiplicity on the destination side. Calculating the matches of \(R\) by starting with \(r\) but using different pivot points cannot produce the same match.

**Proof.** The rule node \(r\) is uniquely determined by the edges \(e_i\) because of the ExactlyOne multiplicity. This means that in a match, exactly one host graph node can be matched to \(r\). However, this means that at least one model item is different in matches started from different pivot points. Therefore, the proposition is always true.

Note that the proposition does not state that the matches do not overlap, it states only that the overlapping part of matches is not the complete match. Using the result of the proposition, we can refine the maximum number of steps required to calculate the match:
\[
\prod_{i=2}^{k} (S(rn_i) - \sum_{j=2}^{i-1} N(rn_i, rn_j)) \times S(rn_1) / c \times h,
\]

(6.15)

where \( k \) is the number of nodes in the rule, \( rn_1 \) is the selected rule node, \( c \) is the number of clients and \( h \) is the quotient of calculating a match locally and sending/receiving a match via the network defined by its pivot point.

### 6.2.3.3 Parallel Matching Based on Pseudo Random Paths

Parallel calculation of matches based on pivot points is efficient if we need all possible matches. However, if we need only the first match, which is common in graph transformations, then the method can be inefficient. The method is particularly slow if there are much less pivot points than the number of clients. This can lead to clients without associated tasks. The reason for the problem is that the previous method is very sensitive against the correct selection of the rule node. This sensitiveness could be eliminated by explicitly selecting the second and other rule nodes similarly, but the condition of unique determination by edges cannot be always satisfied. Another problem is that the network communication is much slower than extending a partial match with a new node. The effect of the network communication is especially remarkable in case of small LHS patterns, where there are a large amount of possible pivot points. The exponential explosion of matching appears in these cases on selecting pivot points explicitly. For example, there is a rule and a host graph, where calculating all matches requires 20ms in the sequential case, while the network communication of sending a partial match is 2ms. Moreover, there are 500 possible primary pivot points and 500 secondary pivot points. This means that we need \( 500^2 \) messages to compute the result. Even if we have ten thousands of clients organized in a perfect hierarchical order, the parallel execution needs more than 20ms. Selecting pivot points for the first rule node only is a natural compromise to solve these issues, although the sensitiveness can lead to problems as mentioned earlier.

A totally new approach is based on allowing completely overlapping matches. This also means that it is possible that the same match is calculated by different clients, which is a waste of clients. However, this waste is not as remarkable if we need only the first match, not all the possible matches. Recall that searching for the first match only is a natural requirement in graph rewriting. By allowing overlapping matches we do not have to select pivot points explicitly. This means that new clients can be dynamically added or removed and only one network message is required that selects which rule to match. Instead of explicit pivot points, the clients try all of the possible pivot points but in random order (Alg. 6.4).

The algorithm is a back-tracking algorithm, which tries to match the nodes of the rule incrementally. Randomization is used in Line 5. This algorithm can reduce the time needed by matching. For example, the starting rule node \( r \) of the LHS can be matched to three host nodes \( h_1, h_2 \) and \( h_3 \). The first client checks matches in the \( h_1 - h_2 - h_3 \) order, while the second client starts checking with \( h_3 \) (and continues with \( h_1 \) and \( h_2 \)). If there is a match in which \( r \) is mapped to \( h_1 \) or \( h_3 \), then the clients executed in parallel can find the match in the first iteration. Nevertheless, the clients can find the match in the second iteration in the worst case. Note that we need to send only the ID of the rule to the clients and one of them sends back
Algorithm 6.4 The Pseudo Random Path algorithm

1: FindMatch(RND, SubMatch)
2: for all RuleNode in GetUnmatchedRuleNodes() do
3:   PivotList= GetPivotsForRuleNode(RuleNode)
4:   while NotEmpty(PivotList) do
5:      SelectedPivot= GetRandomItem(RND, PivotList)
6:      RemoveItem(PivotList, SelectedPivot)
7:      if not ViolatesSubMatch(SubMatch, SelectedPivot) then
8:         AddToSubMatch(SubMatch, SelectedPivot)
9:      if IsCompleteMatch(SubMatch) then
10:         MatchFound
11:         return
12:   else
13:      FindMatch(RND, SubMatch)
14:      RemoveLastMatched(SubMatch)

the complete match. In case of explicitly selected pivot nodes, the network traffic consists of at least two initialization messages (containing the rule ID and the pivot node ID as well) and the result message. In the worst case, the method based on explicitly selected pivot nodes needs a third initialization messages as well.

Note that the pseudo random checking order of rule nodes are used not only on the first level of pivot points, but all further levels as well (due the recursion and back-tracking). This does not mean any additional network cost though, because the order function is calculated in each client locally. Thus, the number of first level pivots does not affect the degree of parallelization possibilities.

Obviously, the order of pivot points must be different in the clients, otherwise they will check the pivot nodes in the same sequence. We use the aforementioned pseudo random function to solve this issue. For each level (for each new rule node), the pseudo random function sets the order in which the pivot points must be checked. Clients use an integer number (parameter RND when initializing the random number generator. This integer is the number of clients started to compute the same rule before.

Proposition 6.6. If the sequential matching algorithm finds a complete match for a rule, then the parallel algorithm finds it as well regardless of the number of clients.

Proof. We show that all clients working in parallel finds to match, if they are not stopped because another client has found it faster. Given an LHS of a rule containing rule nodes r_{1}, r_{2}, \ldots r_{k}.

The client obtains these rule nodes in an arbitrary order. For a given rule node r_{i}, the PivotList is initialized to contain all host nodes that can be matched to r_{i}. These host nodes are enumerated by the while loop (Line 4-13). A host node is removed from the list only if it is checked as well against the actual submatch (Line 7). This means that if we exit the while loop, then we have either a complete match, or we have already checked all of the possible host nodes. Since the same algorithm is used for all r_{i} : 1 \leq i \leq k, thus the client checks all possible match candidates and it can always find a match, when the sequential algorithm has found as well.
In order to be able to compare the approach to the sequential execution, there is a special 0th client in each case. This client is a local (not network) client and it starts checking the pivot points always from the first pivot node, similarly to the sequential approach. This solution ensures that the parallel execution is not slower than the sequential approach with more than the cost of communication with other clients. However, by increasing the number of clients, we increase the chance to find a pivot node combination that is applicable as complete match in less time than the sequential version.

The number of required steps can be expressed as

$$\prod_{i=1}^{k} (S_{T(r_i)} - \sum_{j=1}^{i} N(r_i, r_j)) * R + h * c, \quad (6.16)$$

where $c$ is the number of clients, $h$ is cost of sending the rule ID via the network, while $R$ is a value for which $0 < R <= 1$. $R$ shows the degree of acceleration as a number. Usually, $R$ decreases, if $c$ increases, but currently no exact formula is given to calculate $R$. Future work consists of creating an exact formula that describes the steps without the $R$ function. Until then, we proof by measurements that in practice (6.16) results in less time, than (6.13).

### 6.2.4 Composing the Truly Parallel Graph Transformation Approach

In the previous sections, we have elaborated an algorithm for the transformation-level parallelism and an approach to apply rule-level parallelism. The main drawback of transformation-level parallelism was that it could not handle (could not parallelize) major differences in time required by rewriting the rules. However, the rule-level parallelism offers a method to reduce the time of matching. By composing the approaches, we can obtain a truly-parallel transformation engine that benefits from the advantages of both approaches by supporting both levels of parallelism.

Our truly parallel transformation approach consists of three hierarchy levels (Fig. 6.2). On the first level, there is a unique object, the Master. The Master coordinates the execution of the transformation. Units of the second level are referred to as PrimaryWorkers. A PrimaryWorker is responsible for applying a rewriting rule. Master can have several PrimaryWorkers, it is even possible that PrimaryWorkers connects, or disconnects during the transformation. The third hierarchy level consists of SecondaryWorkers. The SecondaryWorkers compute a match for a rewriting rule by following the pseudo random function discussed earlier. A PrimaryWorker can have several SecondaryWorkers, but a SecondaryWorker cannot be connected to more than one PrimaryWorker at the same time. Transformation-level parallelism is coordinated by Master, the clients are the PrimaryWorkers. Rule-level parallelism is coordinated by PrimaryWorkers, where the clients are the connected SecondaryWorkers.

When a PrimaryWorker is created, it automatically connects to Master and disconnects only when the transformation is finished, or the client is terminated by a user. SecondaryWorkers have a completely different life-cycle. They register themselves first to Master and request a list of PrimaryWorkers. That connection is denoted with dashed lines on the figure. As next, they connect to one of the PrimaryWorkers, but disconnect from there if a match has been found. Then they request the list of PrimaryWorkers again and connect to one of list members.
Chapter 6. Truly Parallel Graph Transformation

Fig. 6.2. The hierarchy of the parallel model transformation approach

The communication between the objects of the transformation engine is based on a distributed algorithm. The following sections elaborate the steps of this algorithm.

6.2.4.1 Initialization and Matching

Firstly, all the participants of the transformation load the definition of the models from the associated model repository. The applications not only load the model definitions, but they initialize several caches, which are used during the calculations. For example, the original transformation definition is converted into a sequence of independence blocks in this phase. Note that the initialization is made offline (i.e. before selecting the concrete input graph), thus only offline algorithms can be applied in this step. In order to minimize overhead during the transformation, we compile the Imperative OCL constraints of the rules as well. The steps of offline initialization are similar on all levels of the hierarchy.

When both the input model and the transformation are selected, the calculation begins. The Master sends a broadcast message of type StartTransformation to the connected PrimaryWorkers and to the registered SecondaryWorkers. Recall that SecondaryWorkers are not connected to any of the PrimaryWorkers at the beginning.

PrimaryWorkers initialize their transformation-specific variables and send a RequestPackage message to the Master. Meanwhile, SecondaryWorkers requests the list of PrimaryWorkers from the Master using a GetPWorkerList function. Masters send the requested list which contains not only the address of PrimaryWorkers, but for each PrimaryWorker the time of the last match request as well. PrimaryWorkers that are running for a longer time are preferred (similarly to [Fatta and Berthold, 2005]). The main reason for this is that the longest running jobs are likely to be among the most complex ones. Recall that each PrimaryWorker has a local SecondaryWorker as well, thus, the match of each rule is computed at least by one SecondaryWorker. However, preferring PrimaryWorkers running for long time means also that highly complex rules are automatically processed by many SecondaryWorkers, thus, the difference between the matching times is reduced. This dynamic load balancing method is very important, because it can compensate the main weakness of transformation-level parallelism.
When the RequestPackage message of the PrimaryWorker arrives to the Master, then it tries to send an awaiting Package to the PrimaryWorker. A Package in this context is a set of data, it is the atom of computation in the transformation-level parallelism, thus, a Package selects a rule of the transformation. If there are no Packages waiting, then the Master obtains the rule from the current independence block, from which Packages have to be created. If the current independence block does not contain an appropriate rule, the Master queries heuristic algorithms (e.g. BidirectionalMCMinimizer algorithm) to get the next rule node to compute. Heuristics are useful in allowing or disallowing the usage of overlapping blocks as well (see StrictHeuristic algorithm). If overlapping blocks are not allowed, or if there are no rule to process, Master suspends the PrimaryWorker, otherwise it sends a InitializeMatch message with the selected Package. The PrimaryWorker broadcasts the selected rule for the connected SecondaryWorkers and waits for a complete match.

Fig. 6.3 shows the messages between the components in a typical case.

6.2.4.2 Finishing the Match, Rewriting

When a SecondaryWorker finds a match, or when it has finished checking all possible pivot nodes, it sends back the result to the PrimaryWorker. PrimaryWorker broadcasts a MatchFound message to all connected SecondaryWorkers (except to the SecondaryWorker that generated the message) to terminate them. SecondaryWorkers disconnect from the PrimaryWorker and requests a new PrimaryWorker list from the Master.

Meanwhile, if the match has failed, PrimaryWorker sends a MatchFailed message to the Master and requests a new Package. If matching was successful, the PrimaryWorker performs

![Fig. 6.3. The sequence diagram of the communication](image_url)
rewriting (executes the associated Imperative OCL code), and obtains the model elements affected by the rewriting. Then, it restores the original state, i.e. the state before rewriting has been applied and requests a new Package.

Recall that heuristics are used to reduce the number of conflicts between the application of rules, however, conflicts may occur. If a PrimaryWorker submits the result of a Package, the Master applies conflict detecting and conflict handling algorithms as described in Section 6.2.2.2. According to the conflict checking results, the Master suspends or allows the submitted rewriting pattern. PrimaryWorkers do not receive information about the acceptance, it is completely handled by the Master. If rewriting of a rule is allowed, Master broadcasts the list of changes to PrimaryWorkers.

Changes in the host graph are not submitted to the SecondaryWorkers by the Master. The reason for this is that SecondaryWorkers may be calculating a match. However, it is essential to synchronize SecondaryWorkers as well. The change lists are numbered by the Master. SecondaryWorkers store the number of last change list applied, while PrimaryWorkers store all change list from the beginning. When a SecondaryWorker connects to a PrimaryWorker, it requests the number of the last change list and synchronizes itself if necessary.

Fig. 6.4 shows the messages between the components in a typical case.

![Fig. 6.4. The sequence diagram of the communication (2)](image)

### 6.2.4.3 Stepping the Transformation

When a Package has been successfully processed (both matching and rewriting), it must be examined whether (i) suspended rules can be allowed, (ii) whether current independence block must be changed and (iii) whether the transformation has finished. The first examination is
Chapter 6. Truly Parallel Graph Transformation

simple, it is based on constructs described in Section 6.2.2.2. An independence block is finished when all rules inside the block have been finished. If a rule is marked as exhaustive, then it is finished only if the last match has failed (the pattern cannot be matched in the host graph anymore), otherwise the rule is finished after it has successfully rewritten, or an unsuccessful match has been retrieved. The whole transformation finishes if all rules of the last independence block have been finished.

6.2.5 Performance Measurements

We have implemented a model transformation framework that supports the parallel execution of transformations. Both transformation-level and rule-level parallelism are supported based on the solution elaborated in the previous sections. Implementation details are discussed in more detail in Chapter 7. After creating the transformation framework, we have recognized three characteristic properties of the transformation that can be critical in case of parallel execution. Moreover, we have created a reconfigurable case study to apply the tests and measurements on. This section introduces the performance factors, the case study and the results of the measurements.

6.2.5.1 Performance Factors

The performance of parallel transformation is obviously affected by the underlying network. The communication overhead and the number of messages generated during the transformation have been minimized during the evolution of our approach, but still, messages are sent over the network. This means that the presented approach may not be efficient, if the transformation consists of many simple rules. Simple means in this context that the time of rewriting is in the order of magnitude of sending Packages over the network.

The performance of parallel execution is also affected by the ratio of minimum - maximum time of rule rewriting (referred to as complexity ratio). Complexity ratio especially applies to transformation-level parallel algorithms. If there is a rule which requires the 90% of the complete time of execution, transformation-level parallelism cannot accelerate more than 10%.

The third performance factor is the degree of dependence between the rules. A transformation without metaconflicts can be applied fully in parallel. Model transformations, especially code generator transformation have the tendency to satisfy this condition.

6.2.5.2 The Case Study

Our case study models a local network of a factory. More precisely a modern assembly line of a computer factory. The assembly line consists of several components, which are either common desktop computers, or special devices. Both computers and other devices can have several subparts (for instance peripheries, IO cards or connected sensors). The case study model can describe the full hierarchy of this highly complex system. The model describes the communication network between the components of the assembly line and the exact structure inside a computer/device (the configuration of subparts). The aim of the case study model is to give an overview of the system in a graphical form.
Moreover, we use model transformations to find problematic configurations in the queue. A problematic configuration means in this case that a special combination of computers, devices and subparts causes errors. (For example, if there are two computers with a special type of network cards, then the communication between these computers can slow down to an unacceptable speed.) A problematic configuration can consist of any number of computers, devices, or subparts. The model transformation method that we use, can find these configurations and replace the conflicting components. Fig. 6.5 shows an example model, where the components of the problematic configuration are denoted by ellipses.

We have created a test generator that can simulate the case study. The application generates a model transformation (including the rules), a metamodel for the transformation and a host model. Rules in this case describe a problematic configuration (LHS) and its replacement (RHS).

We have added false matches to the host model, namely nodes and edges to construct a sub-match, but not a complete match. This way we can simulate similar, but not exact configurations that would be common in a real life example.

The generator application has several input parameters: (i) number of rules (ii) number of nodes/edges in rules, (iii) average number of false matches in the host model, (iv) average number of metaconflicts between two rules.

The test results can be customized by setting these input parameters. We have applied several tests by varying these properties in order to test the performance.

### 6.2.5.3 Test Results

We measured the time of transformations in case of sequential rewriting and parallel rewriting with one to three PrimaryWorkers and varying number of SecondaryWorkers. The exact number of client computers is discussed in each case separately.

The tests were executed on a local network with 100Mbit bandwidth. In communication, we used a modified version of the UDP protocol that offers reliable delivered messages (Section
7.1.3.2). Client computers have a 2.1Ghz Pentium Core2 Duo processor with 2Gb of RAM, the operation system was Windows XP.

The results of the performance measurements are shown in the tables below the pictures. The cells of the table show the relative time needed by the transformation, where the sequential transformation requires 100 "time-unit" in all of the cases. The rows of the table contain results for a given configuration, while columns show the results of different configurations for a given model.

### Network Overhead

The first performance factor is stressed in Table 6.1. Only the local (0th) SecondaryWorker was used in these tests. The first row shows the relative complexity of the cases, where the first number is the number of rules in the transformation, while the second number is the time of execution in the sequential case (in ms). As the table shows, the difference between the efficiency of 20/400 and 12/17000 is almost negligible. This means that the network overhead is unremarkable even if the average time required by a single rewriting rule is approximately 20 ms. Although the parallel execution has a significant overhead if we use only one PrimaryWorker, this overhead disappears when we increase the number of clients (PrimaryWorkers).

<table>
<thead>
<tr>
<th>Complexity Ratio</th>
<th>20/400</th>
<th>100/2300</th>
<th>12/17000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PWorker</td>
<td>150</td>
<td>131</td>
<td>109</td>
</tr>
<tr>
<td>2 PWorker</td>
<td>70</td>
<td>67</td>
<td>63</td>
</tr>
<tr>
<td>3 PWorker</td>
<td>53</td>
<td>49</td>
<td>47</td>
</tr>
</tbody>
</table>

**Table 6.1.** Performance factors - Network overhead

### Complexity Ratio

The second performance factor is analyzed in Table 6.2 and in Table 6.3. The first table shows results if only the transformation-level parallelism is used, while the second table contains results created with several SecondaryWorkers (exact number is shown in the first column). In both cases, the first row shows how much time the most complex rules of the transformation require in comparison with an average rule (the **complexity ratio**). For example, 30*10* means that the transformation has many rules applicable in x ms, but there are two rules, which need 30x ms and 10x ms respectively.
According to these results, it seems that complexity ratio can strongly limit the performance if rule-level parallelism is disabled. Moreover, the results show that the performance is affected not only by the complex rule itself, but the position of the problematic rule in the transformation as well. That is why the parallelization of 15* is not as efficient, than the parallelization of 30*/10*.

<table>
<thead>
<tr>
<th></th>
<th>15*</th>
<th>30*/10*</th>
<th>10*</th>
<th>3*/3*/3*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PWorker</td>
<td>104</td>
<td>110</td>
<td>104</td>
<td>107</td>
</tr>
<tr>
<td>2 PWorker</td>
<td>92</td>
<td>79</td>
<td>81</td>
<td>56</td>
</tr>
<tr>
<td>3 PWorker</td>
<td>92</td>
<td>77</td>
<td>73</td>
<td>37</td>
</tr>
</tbody>
</table>

**Table 6.2. Performance factors - complexity ratio**

The results in Table 6.3 clearly indicate that rule-level parallelism is an efficient way to eliminate inhomogeneous matching time. Even if the pseudo random function is hard to be described by a well-defined formula, increasing the number of SecondaryWorkers reduces the time required by the transformation. However, as Table 6.3 shows, the efficiency gained from using rule-level parallelism is less than in case of transformation-level parallelism. For example, we need 25 SecondaryWorkers to reduce the time of the transformation to 67% of the original (case 15*). However, this 67% is still much better than the 92% produced by two SecondaryWorkers.

<table>
<thead>
<tr>
<th></th>
<th>15*</th>
<th>30*/10*</th>
<th>10*</th>
<th>3*/3*/3*</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 PW+2 SW</td>
<td>92</td>
<td>79</td>
<td>81</td>
<td>56</td>
</tr>
<tr>
<td>2 PW+10 SW</td>
<td>81</td>
<td>70</td>
<td>63</td>
<td>41</td>
</tr>
<tr>
<td>2 PW+25 SW</td>
<td>67</td>
<td>54</td>
<td>47</td>
<td>33</td>
</tr>
</tbody>
</table>

**Table 6.3. Performance factors - complexity ratio 2**
The reason for the difference in performance gain is that matches cannot be parallelized as efficient as rules. This topic is discussed in more detail in Section 8.3.

Metaconflicts

Third performance factor (Table 6.4) focuses on the number of metaconflict relations in the transformation (shown in the first row). No additional SecondaryWorkers were used (only the 0th). In all of the four cases, the original transformation contains 12 rules. However, it is able to create more, than 12 metaconflicts between the rules, because the rule consists of many nodes and each of these nodes can lead to a metaconflict. According to the results, metaconflicts do not limit the performance unless the transformation contains many crosscutting metaconflicts (MC60). Note that between independence blocks, the BidirectionalMCMinimizer algorithm is used by default.

![Graph of metaconflicts](image)

<table>
<thead>
<tr>
<th></th>
<th>MC60</th>
<th>MC24</th>
<th>MC12</th>
<th>MC3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Client</td>
<td>106</td>
<td>105</td>
<td>107</td>
<td>103</td>
</tr>
<tr>
<td>2 Client</td>
<td>73</td>
<td>59</td>
<td>54</td>
<td>53</td>
</tr>
<tr>
<td>3 Client</td>
<td>66</td>
<td>37</td>
<td>37</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 6.4. Performance factors - metaconflicts

6.3 Chapter Summary

Graph rewriting-based model transformations are popular in model-driven approaches, because they are flexible, they have a mathematical background, and they can be defined graphically. However, the performance of graph rewriting is not always enough in case of complex input models and transformations. Even applying a single rewriting rule is an NP complete problem. There are different techniques to accelerate the transformations for example by searching plans, but these techniques support only a small subset of graph transformations. However, the parallel execution of transformation is an efficient way to solve the performance issues.

The chapter has presented a truly-parallel model transformation approach. The approach consists of transformation-level and rule-level algorithms, which allows using parallelism in applying different rules and in computing matching respectively.

The idea behind the transformation-level parallelism is to analyze the steps offline and try to avoid the dependent rules. Firstly, a construct has been given that creates blocks of transfor-
mation steps, where rules of a block are always applicable in parallel. Secondly, the approach is extended by allowing the overlapping execution of the blocks. Heuristics have been defined to avoid conflicts between the rules, and a conflict handling algorithm has been presented to resolve conflicts if they occur.

Two algorithms have been presented to support rule-level parallelization. The first algorithm selects a model item from the rule definition and tries to check the associable host model items in parallel. The second algorithm generalizes this idea, and the clients check the possible host nodes in randomized order.

The transformation-level and rule-level algorithms have been summarized in the truly parallel transformation approach. It has been presented how this approach coordinates and synchronizes the algorithms.

The practical relevance of the parallel execution has been illustrated by a case study with respect to three recognized performance factors. According to the results, it can be seen that the parallel model transformation is not only a theoretical construct, but a practical approach with considerable performance factors.
Chapter 7

Application of the Results

The results of my research, namely the (i) n-layer attribute handling, (ii) concrete syntax definition, (iii) optimized, metamodel-based constraint compiler, and (iv) the truly parallel model transformation engine – presented in Chapter 4, 5 and 6 – form an efficient, transformation-based approach to model and transform visual languages. These techniques have been successfully applied in many industrial and research-oriented applications, such as resource editors for mobile phones, supporting generative techniques, or modeling network communication. This chapter presents a practical validation of the mainly theoretical results presented earlier. We show that these results are not only useful in theory, but in real-world application as well.

The theoretical results have been realized in the Visual Modeling and Transformation System (VMTS). In this chapter, an overview is given on the basic concepts of VMTS (Section 7.1), which is followed by the detailed description of the components based on the result of my research (Section 7.2). At the end of the chapter, we elaborate some of the most remarkable real-world case studies based on the results.

7.1 The Visual Modeling and Transformation System

Visual Modeling and Transformation System (VMTS) is an n-layer metamodeling and model transformation environment. VMTS benefits from the results of the mathematical background of formal languages, graph theory, category theory, graph rewriting and metamodel-based software model transformation. VMTS is an approach that uniformly treats model storage and model transformation. Moreover, VMTS has built-in support for several visual languages including UML diagrams, feature models, and also provides facilities to create custom domain-specific languages simply. VMTS consists of several components as shown in Fig. 7.1.

Attributed Graph Architecture Supporting Interface (AGSI) offers a high-level graph interface for the other components to reach the underlying data repository. VMTS supports two kinds of data repository: (i) AGSIDb that is based on a Relational Database Management System (RDBMS) and (ii) AGSI Compact Framework that uses a custom, file-based model factory. AGSI is a unified interface though, thus, different kinds of repositories can be reached transparently.
VMTS Presentation Framework (VPF) is solution to support creating, displaying and editing the models in a graphical environment. Adaptive Modeler is an application based on the VPF, it provides an easy-to-use user interface for the core functions of VMTS.

Model transformation is applied using Rewriting Engine and Visual Model Processors (VMPs). Rewriting Engine implements the transformation engine, while VMPs define the transformation steps using VMTS Control Flow Language [Lengyel et al., 2006e] and graph rewriting rules in a graphical form.

VMTS offers another built-in way to process models: Traversing Processors are used for traversing the models in order to generate program code or other artifacts. By using Traversing Processors, we can generate a high level interface to query and modify the models.

Constraints in modeling and model transformations are compiled to a validating binary by the OCL Compiler component. This binary is used every time, when validation is required. Constraint optimization is automatically, transparently used in the compiler.

To validate the theoretical results, Attribute panel, AGSI Compact Framework, VMTS Presentation Framework, VMTS Presentation DSL, OCL Compiler and the AGSI Parallel Transformation Engine were implemented by the author.

7.1.1 Modeling Framework

In order to define and handle visual languages, it is essential to have a flexible modeling framework. One of the main practical contributions presented in this work is the VMTS Presentation Framework that is an editor for visual languages. It implements the theoretical results in n-layer attribute algebra and it supports defining the concrete syntax in a graphical way.

7.1.1.1 VMTS Presentation Framework

The Presentation Framework has metamodel-variant and metamodel-invariant parts. The invariant parts are capable of supporting a model with an arbitrary metamodel. Fig. 7.2 shows the main types of invariant parts: (i) the tree representation of the model and the visualization structure, (ii) the attribute panel, which displays the metamodel-defined data implementing the
Chapter 7. Application of the Results

n-layer attribute algebra and the visualization information, and (iii) the toolbar, which contains the model elements that can be dropped on the canvases.

Besides the default presentation, the attribute panel offers a custom editor interface for special attributes (referred to as VMTS Custom Property Editors), such as file saving, or code editing. VPF has several built-in editors for the most often used attribute types. VPD plugin, the plugin used in creating the concrete syntax relies heavily on these custom editors.

The toolbar enlists the non-abstract model elements from the metamodel. The user can drag an element from this list, and drop it onto a canvas. Although this representation is appropriate in most cases, it is not elegant e.g. when the UML object diagram is displayed. In this case, the metamodel is a UML class diagram possibly with many classes and associations. The traditional approach is to display an Object and a Link shape, and the user selects the desired type after dropping them onto the canvas. Generality is not restricted, since the toolbar should work for models with arbitrary metamodels. We defined a \textit{MetaMetaBased} mode for the toolbar, which means that the control displays one element for the model elements having the same meta-metamodel element. This is exactly the behavior that we wanted for the object diagrams, since only one item is displayed for all objects (since their meta-metamodel element is the same: metaclass), and the associations are treated similarly. If an object or a link is dropped on the canvas, a pop-up list appears and offers the available model elements in the metamodel.

The variable part of VPF uses plugins to offer individual visualization and editing features (Fig. 7.3). The framework offers a core functionality, and it can be extended by the model-specific plugins. A plugin is always attached to a metamodel whose models it supports (e.g. UML statechart metamodel for the statechart plugin). On loading a model, if no plugin assigned to the metamodel was found, then a default plugin is used, which is referred to as Abstract Syntax Plugin (Fig. 7.3 c). This general environment is also used for editing metamodels on the uppermost modeling layer. The Abstract Syntax Plugin is a part of the framework.

One of the most crucial part of a modeling framework is the organization of the models. Requirements such as supporting model aspects and multiple views of the same elements need flexibility in the diagram structure. In VPF, canvases can be defined for the models, which display a part of the system, typically, an aspect of the model. Each model can have multiple...
canvases. Since a canvas can be considered an aspect or a view of the model, it is a natural requirement that a model element can appear on more than one canvases, possibly with different visualization properties. The framework supports multiple presentation and customized visualization of the model elements. Furthermore, it synchronizes the same elements on different canvases by using the Model-View-Controller (MVC) design pattern [Gamma et al., 1995]. In VPF, each model item has exactly one Model (that represent the attributes, for example) and one View-Controller pair for each canvas. Note that the synchronization between the canvases is necessary, since the simultaneous presentation of different canvases can require automatic refresh of the common properties on all the canvases.

VPF offers built-in features and services that are used frequently in the visualization of a model element (e.g. resizing, relocation or docking). To offer these services, the framework contains generic base classes. When developing a plugin, one needs to derive from these classes and define the customized behavior or appearance only. VPF has built-in support for creating of model items, automatic saving and loading; supporting containment with drag and drop, event handling, drawing and accessing helper properties on visualization information.

N-layer attributes

Attributes in VMTS are based on the n-layer attribute algebra as described in Section 4.2. This also means that the instantiation is layer-transparent and there is only one instantiation transformation. In AGS-IDB, attributes are represented by XML documents, while in AGSI Compact Framework, they are described in an object oriented hierarchy. Both approaches handle attributes in a tree-structure. The attribute panel of VMTS Presentation Framework visualizes the attributes in a property grid in both approaches. The attribute structure of the metamodel level is automatically transformed into a validation scheme for the instance level models by the framework. In case of AGS-IDB, this scheme is an XSD document – the Schema XSD – that is used to validate the attributes of the instance level during modeling [Mezei et al., 2005f]. In AGSI Compact Framework, we use an object template with similar functionality.

Dual Container Hierarchy

There are several model types that require containment information with drag and drop support between the elements. In our approach, the containment is handled between nodes only; the edges cannot contain other model-elements. The elements in a container are ordered, similarly to the windows of the applications in a graphical window system. Obviously, the order can be changed by the users during editing the diagrams. In the containment hierarchy, each model
element must have exactly one container. This uniqueness is necessary, because model-processing algorithms (e.g. traversing model processors, code generators, model transformations) are much simpler and faster if this condition succeeds. Therefore, VPF defines a containment chain between the Model objects. There are two variables in each model object that supply the connection between the elements of the hierarchy: Children and Container. These two properties facilitate the simple navigation in the containment hierarchy. The containment relationship between the model elements follows the rules of the metamodel. The list of the children is an ordered list, the position in the list is used for the hierarchical and visual ordering.

Although the containment hierarchy defined by the metamodel is more or less straightforward, the issue needs an additional mechanism, when several views and the modification of the order are considered. The model elements on the canvas depend only on the user, i.e. which elements are placed on a canvas by using the drag and drop functions. Fig. 7.4 shows an example. Fig. 7.4a represents the simplified steps of a phone call. Major steps are focused, minor steps are hidden. Fig. 7.4b shows the details of the Dialing state (darker color). In the first case the container of the Dialing state is the Active state, but in the second case the canvas is the container. Moreover, these hierarchies are compatible with the hierarchy defined in the metamodel, but not always identical.

On the one hand the concept of the unique container should be maintained, on the other hand a model element can have different containers on different canvases. The hierarchy defined by the metamodel is followed by the hierarchy of the Model components, however, the actual order on a canvas must be recorded in all of the canvas-dependent components. Since this hierarchy is closely related to the event handling mechanism, which is the responsibility of the Controller (as described in [Gamma et al., 1995]), a Controller-based hierarchy is established along with the Model-based hierarchy. The root of the Controller-based hierarchy is the Controller component of the canvas. By using this dual containment hierarchy, the real topology and the visualization have successfully been separated.

In the Controller-based hierarchy, we distinguish reference controllers and non-reference controllers. Changes made on the first type of controllers affect the Model-based hierarchy as well. Consistency between the hierarchies is automatically ensured by the framework. VPF visualizes the different hierarchies in two tree controls. The tree controls are always synchronized with the canvases.
Persistence and behavior

VPF uses the Model-based containment hierarchy for saving the model elements. Firstly, the model itself is saved, then the canvases. The model elements are saved using a depth-first search. The loading algorithm also uses the containment chain. Firstly, the model is loaded, secondly the canvases. Loading the model is performed in several steps: (i) Nodes are loaded (using depth-first-search), (ii) the Model-based containment is restored for nodes. (iii) Edges are loaded, and (iv) the Model-based containment is restored for edges. (v) Finally, the Controller-based containment hierarchies are restored for each model-element: both edges and nodes.

As far as the behavioral aspects are concerned, the Controller-based hierarchy is appropriate for event handling. Recall that the root of this hierarchy is always the Controller component of the canvas. For instance, if the user clicks on a state, the event is delegated to the active canvas by the execution environment, and the message is passed to the Controller component of the selected state. The propagation of the events are not obvious for overlapping elements. If the control contains elements, it forwards the event to its children with a depth-first-search algorithm. If a container and the contained item both can handle the event, then it is handled by the contained item.

The behavioral commonalities allow VPF to provide implementation for the behavior of model elements. The Controller objects in the framework have an attribute State that can change according to the user actions. For each Controller class there exists a matrix with the transition between the different states. The event-handling (e.g. a model element can be moved only if it is in the Selected state) and the visualization (e.g. the selected items should display a selection border) are based on the State attribute. The events sent by the execution environment can be refined by the framework to provide more customized notifications for the model elements.

7.1.1.2 Visual Appearance Definition

Metamodel does not describe the presentation of a visual language. Section 4.3 has elaborated a solution based on a special domain-specific language, the VMTS Presentation DSL (VPD). In VMTS, creating and using concrete syntax consist of five main steps. These steps are shown in Fig. 7.5. (i) Presentation Framework instantiates VPD Metamodel, a new VPD Model is created. To facilitate the definition of the concrete syntax, a plug-in (the VPD Plugin) was implemented based on VMTS Presentation Framework. (ii) To improve the effectiveness, there is a support for processing VPD Models (the concrete syntax definitions) automatically, using model transformation techniques. (iii) The transformation converts VPD Models to CodeDOM [Thai and Lam, 2003] models. CodeDOM is an abstract code representation. (iv) From the CodeDOM model, source code is generated with the .NET CodeDOM technology. (v) The source code implements a plug-in that can be used directly in VMTS Presentation Framework to edit the models of the domain.

VPD Plugin offers an easy way to create the weaving contexts, namely the mapping between the topological definition and the concrete syntax. Using the selected topological definition (metamodel), weaving contexts can be created automatically by the plugin. The automation
The key feature in defining Appearance Definitions is the ability to define VMTS Custom Property Editors introduced in the previous section. VPD Plugin defines a custom attribute editor – Appearance Editor – for editing the appearance of the shapes. Appearance Editor (Fig. 7.6) is an image editing form with support for modeling the primitives of the regions. Recall that primitives are simple graphical objects (e.g. rectangle, circle, line) used to define a region. Moreover, a region is responsible for visualizing a part of the model item, or the model item itself.

Primitives can be created using drag-and-drop actions, they can be resized and moved, or changed (for example points can be added to or removed from splines). Every primitive supports a large amount of customizable attributes, such as line style, or brush mode. Properties of the primitives can also be edited simply, since they appear in the property grid residing on the right hand side of the form. Appearance Editor has additional facilities such as zooming, support for undo/redo operations, or layer management.
Chapter 7. Application of the Results

7.1.2 Extending, Optimizing Constraint Handling

The information conveyed by a model created by a traditionally generic modeling language has a tendency to be imprecise [Warmer and Kleppe, 2003]. However, it is able to attach textual languages to visual model definitions using the OCL language. This section introduces the OCL compiler implemented in VMTS. The compiler supports domain-specific constraints as well as optimizing constraint validation.

7.1.2.1 OCL Compiler

Fig. 7.7 shows the main steps of constraint compilation in the OCL Compiler of VMTS. Firstly, the user defines the constraints in OCL, secondly the constraint definitions are tokenized and syntactically analyzed. The lexical analysis reads the constraint definition as a text and creates a sequence of tokens from it. Tokenization is performed by Flex [Flex, Official Homepage, 2007]. Syntactic analysis builds a syntax tree using the grammar rules specified in OCL. The grammar cannot be mapped directly to the OCL specification because of its ambiguities. The grammar rules are simplified to solve this problem. The simplified information is reconstructed in the later compilation steps, where we have more information to perform the analysis (e.g. about available types and defined variables). Syntactic analysis is supported by Bison [Bison, Official Homepage, 2007].

As next, the syntax tree is extended with type information, and implicit self references in the semantic analysis phase. The result is the semantically analyzed syntax tree. The semantic analysis reconstructs the simplification made in the grammar in the syntactical analysis phase.

In the next step, the constructed and semantically analyzed tree is transformed to a CodeDOM tree, then the compiler transforms the CodeDOM tree to C# source code. Finally, the compiler compiles the source code and builds the executable. The output of the OCL compiler is a .NET assembly (a .dll file) that implements the validation on the OCL constraint.

OCL optimization features are implemented in the core of the OCL compiler, thus, constraint relocation w.r.t. to the multiplicities and caching are automatically handled by the compiler. The optimization algorithms strictly follow the theoretical algorithms presented in Section 5.3. According to our tests, optimization can reduce the number of model queries by 10-15%.
7.1.2.2 Compiler Extensions

In VMTS, we have created a class library (referred to as OCL class library) that contains classes for the base types defined in OCL, for example classes for the types Set, Boolean, or Tuple. The generated code highly relies on these classes: the behavior and the supported functions of the OCL types are implemented here. Handling constraint expressions based on a class library has two main advantages. (i) In the semantic analysis phase, type checking can use the built-in types and can easily check whether the requested operation is supported. The basis of the type-checking is the metamodel definition. (ii) Accessing the underlying model can use a common interface (ModelInterface). Fig. 7.8 shows an overview about the classes of ModelInterface used in VMTS.

ModelInterface hides the underlying representation of the model items, thus, it grants access to the model items for the compiler in a transparent way. ModelInterface and the OCL class library are the key features in extending the original OCL compiler. Note that the transparency achieved by ModelInterface can grant reusability of the presented solution in other metamodeling systems as well.

The ModelInterface and the OCL class library makes it possible to handle complex attributes of the n-layer attribute algebra easily. We have added a class ModelAttribute to the class library in order to support navigation between the subattributes as described in Section 5.2.1. ModelInterface allows handling attributes of modeling edges as well. Attributed edges are handled as an edge without attributes and an association node containing the attributes. This representation is compatible with the original definition of OCL and thus, it is easily applicable by ModelInterface.

Constraints are useful not only in modeling, but in specification of model transformation rules, especially in validated model transformations [Lengyel, 2006]. OCL compiler of VMTS has been extended to support transformation-specific features as well [Mezei et al., 2006i]. For example, a constraint expression can have more than one associated metamodel to check the model types in. This is a common need in transformation rules, because a rule can have several...
output models. For instance, a code generator rule modifies the source model and generates the CodeDOM model representing the result at the same time. VMTS OCL compiler supports this feature with the help of ModelInterface.

The basic, optimizing OCL compiler with metamodel support serves also as the basis of several current research fields, such as describing the connection between the LHS and the RHS of a rule by using Imperative OCL, which is a dialect of OCL.

### 7.1.3 Model Transformation

Efficient model transformations are essential in model-based approaches. The transformation engine of VMTS had been developed before my research began. However, the results presented in Chapter 6 added a completely new viewpoint, a completely new way to accelerate transformations by using a computer network in calculations.

#### 7.1.3.1 AGSI Compact Framework

Before focusing on parallel model transformations, we have compared the performance of VMTS with the performance of other model transformation tools. AGSI Compact Framework did not exist at the time. The basis of the comparison were the case studies of [Varró et al., 2005]. According to the results, AGSIDB was very slow because of the underlying relational database. Therefore, we have created the AGSI Compact Framework, which can load models to memory and apply rewriting on an in-memory model factory. To hide the differences between the AGSI implementations, we have created the common interface referred to as AGSI. We have applied performance tests on the AGSI Compact Framework as well, which has shown that in-memory model handling requires approximately three order of magnitude less time to apply the same transformation as AGSIDB. The main reason for this huge difference is that Compact Framework uses memory pointers, while AGSIDB uses database calls. Furthermore, the tests have shown that VMTS (based on the AGSI Compact Framework) is one of the fastest transformation engines.

Table 7.1 shows the time required by matching a specific pattern in different transformation tools. Note that the model and the transformation is borrowed from [Varró et al., 2005], where we have used the case of long transformations (long TS), without optimization. The results are measured in ms, for a single application of the rules. Note that neither the transformation, nor the transformation engine settings were optimized during measurements in VMTS.

<table>
<thead>
<tr>
<th>Model size</th>
<th>AGG</th>
<th>PROGRES</th>
<th>Fujaba</th>
<th>DB-based</th>
<th>VMTS\textsubscript{DB}</th>
<th>VMTS\textsubscript{Compact}</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1.86</td>
<td>0.62</td>
<td>0.15</td>
<td>4.15</td>
<td>1326</td>
<td>0.21</td>
</tr>
<tr>
<td>5001</td>
<td>1116.34</td>
<td>269.58</td>
<td>0.26</td>
<td>20.47</td>
<td>3 \times 10^4</td>
<td>1.2</td>
</tr>
</tbody>
</table>

**Table 7.1.** Performance of model transformation approaches

AGSI Compact Framework is implemented as a class library. It has classes for each fundamental type of model items of VMTS (e.g. nodes, edges, models). It also supports initializing the model factory from a database using the AGSIDB component. The Compact Framework
uses a similar data representation, as AGSIDB and exactly mirrors the relations between the model items in the database. Differences between the data representations are: (i) model items are objects, instead of table rows, and (ii) attributes are described by hierarchically nested objects instead of XML documents. Moreover, Compact Framework supports caching the relations between the model items according to several aspects. For example, a node has not only a list with the connected edges, but has several hashtables indexed by the type of edges. This means that if we want to find a connected inheritance edge, then we have to check only the hashtable of inheritance edges. Caching hashtables are automatically created and maintained by the framework.

Compact Framework supports serializing and deserializing the models into and from model factory files. The file structure, used in serialization is similar to XML, but it is more compact and it can be loaded (parsed) much faster. Pointers to model items or attributes can be serialized and restored as well by using their unique ID.

AGSI Compact Framework does not support all features of AGSIDB currently. For example, Compact Framework does not support handling the presentation data of model items currently, because the primary aim of Compact Framework was to accelerate model transformation not modeling. This feature is part of our future work (Section 8.3).

7.1.3.2 VMTS Parallel Transformation Engine

Because of the large difference between the performance of AGSIDB and AGSI Compact Framework, parallel transformation engine has been implemented using the later component. The components of the theoretical parallel transformation engine (Master, PrimaryWorker and SecondaryWorker) have been implemented in different classes. These classes do not define the exact communication protocol, but use messaging primitives of a common interface, the IParallelCommunicator interface. Objects implementing this interface are used as a communication bridge between the main components of the parallel transformation framework. A typical hierarchy is shown in Fig. 7.9.

![Typical Component Hierarchy in VMTS Parallel Transformation Engine](image-url)

Fig. 7.9. Typical Component Hierarchy in VMTS Parallel Transformation Engine
Recall that there is always exactly one Master coordinating the execution on the transformation level. PrimaryWorkers receive rules to be applied from the Master, and they coordinate rule-level parallelism. SecondaryWorkers are used to find a match for a rule in the host graph.

The main purpose to separate IParallelCommunicator from the Master, PrimaryWorker and SecondaryWorker is to make computation independent from the underlying network protocols (e.g. TCP/IP, or UDP). Note that this also means that message lost in unreliable protocols, such as UDP, can also be handled by the objects implementing the interface. The current implementation and test results presented in Section 6.2.5 are using a self-developed quasi reliable UDP protocol (it grants that all packages will be delivered and the order of the packages does not change). If the flexibility in choosing the network protocol is not required, then the base components can be unified with the interface.

Both the Master and the workers have a configuration file, which is loaded when the hosting application is loaded. Data storages are initialized and network communication is established in the associated IParallelCommunicator components according to the settings. The Master does not know the address of the Primary-, or SecondaryWorkers when it starts, because the list of available workers can change dynamically at run-time. When a worker sends a register signal to the Master, then the Master stores the ID of the worker in a list. This ID is a globally unique identifier (GUID) of the client. Master differentiates between PrimaryWorkers and SecondaryWorkers at registration. Communicator object attached to the Master also stores the worker ID and maps it to the network address of the worker. Thus, it supports automatic translation between the IDs and the network address.

Messages used in communication have a predefined scheme. More precisely, there are two schemes: one sent by the clients and one sent to them. Client in this context means a subordinate component, e.g. a PrimaryWorker is the client of the Master, while a SecondaryWorker is a client of a PrimaryWorker.

Messages sent to clients are structured as follows: (i) the first byte identifies the type of the message, which is (ii) followed by the serialized message itself as a byte array. This message is usually an ID of a rule, a complete match, or the change list generated by a rewriting. Messages sent by the clients have similar structures: (i) the first byte identifies the type of the message, it is (ii) followed by the GUID of the sender client and (iii) by the message itself. However, the presented structures can be overridden by communicator objects. In case of reliable UDP protocol, message type is followed by 4 bytes describing the ordinal number of the message. The lack of passing special data types e.g. object references in the messages makes communication slower, but grants that the underlying network protocol can be changed easily.

Another issue in creating memory-based parallel execution is to support rollback. Recall that the PrimaryWorkers apply rewriting of rules, but have to restore their original state after sending the change list to Master. This issue is handled by a modified version of Command design pattern. Each atomic change made by the Imperative OCL, is noticed as a change command, when applying the rule. Supported atomic changes are for example CreateItem, or SetAttribute. Each Command has an Apply and a Restore method, which are used in applying or restoring the changes made by rewriting. Moreover, each Command can be serialized or deserialized, which simplifies sending changes from the PrimaryWorkers to the Master.
7.2 Model-Based Development with VMTS

In this section, case studies are provided to prove the applicability of the presented methods. The first case study shows in a nutshell how a special domain, namely, UML Statechart diagrams can be described by n-layer metamodels and VMTS Presentation Framework. The second case study elaborates the creation of concrete syntax for one of the most well-known visual programming languages the FlowChart diagram. The third case study shows industrial examples, where concrete syntax definition has been used. The fourth case study presents a simple model and introduces the efficiency of constraint optimization on this model. The fifth part of the section is a presentation about other industrial fields, where the elaborated theoretical and practical results have been successfully applied.

7.2.1 Support for UML Diagrams

UML is one of the most well-accepted general modeling language family. In order to prove that the construction of visual languages based on metamodeling is efficient, it is worth showing that UML modeling languages can be described by metamodeling. Although, UML has a metamodel-based definition, it contains circular dependencies and another constructs, which makes impossible to use this metamodel definition in VMTS. However, we can create our own metamodel for each UML language.

The metamodel of UML statechart diagram is shown in Fig 7.10/a. Each model element are inherited from StateElement, except the Region. StateElement is an abstract base type, it does not appear on the instance level. Region is a helper model item to express parallel regions in the state. Other model items describe elements of the language according to the specification of Statechart diagram [UML 2.0, 2006]. Transitions between the elements are modeled by an edge from and to StateElement. This metamodel can be easily constructed using the built-in Abstract Syntax Plugin of VPF. Note that both the topology and the attributes are defined in a graphical form, no program code is required.

![Fig. 7.10. The Statechart Metamodel and an example model](image)
To support customized visualization of Statechart diagrams, we have created a Statechart plugin based on the features of VPF. Fig. 7.10/b shows an example model visualized by the plugin. Recall, that instance level attributes are automatically forced to follow the attribute structure created in the metamodel.

We have created metamodels and visualizing plugins for other UML language as well (Fig. 7.11) including Use Case diagram (a), Deployment diagram (b), Activity diagram (c), Class and Object diagrams and Sequence diagram (d).

![Fig. 7.11. UML models in VMTS](image)

### 7.2.2 Graphical Programming Languages

Defining the concrete syntax of visual languages is simple by using VMTS Presentation DSL and VPD Plugin. The concrete syntax of *FlowChart* diagrams can be constructed as follows:

1. The structural definition, i.e. the metamodel of the visual language is created.
2. A new VPD model (a new concrete syntax definition) is created.
3. By using the built-in mapping feature of VPD plugin, Weaving Contexts are generated automatically from the metamodel.
4. The appearance is defined for the language elements.
5. VPD Transformation is used to create a plugin from the VPD model.
6. The generated plugin can be used in editing the models of the target domain.

Appearance is defined by a graphical editor. This editor visualizes the content of the attribute describing the appearance, thus, instead of writing the appearance definition in textual form (e.g. in XML), it is able to edit it visually. We have used VMTS Presentation DSL to implement the plugin of Flowchart diagrams (Fig. 7.12/a). We have compared the time required by implementing the flowchart plugin to create it by the VMTS Presentation DSL. The implementation required 3 days, while visual concrete syntax definition needed only 5 hours. We have
also implemented another popular visual programming language, the Nessie-Schneidermann diagrams with the same techniques (Fig. 7.12/b). The creation of visual definition needed similar amount of time.

7.2.3 Network Modeling

Concrete syntax definition is useful not only in creating visual programming languages, but in industrial case studies as well. In the followings, three network-based DSL is presented, for which the visualization have been defined by VMTS Presentation DSL.

Modeling GSM networks

Graphical representation of GSM network components and their hierarchy is a natural need. Visualization can simplify finding topological problems in the network. GSM Metamodel (Fig. 7.13/a) contains only the most common components of GSM networks: Gateway Mobile Switching Centre (GMSC), Home Location Register (HLR), Visitor Location Register (VLR), Mobile Switching Center (MSC), Base Station Controller (BSC), and Base Transceiver Station (BTS). To emphasize the similarities, the common attributes in HLR and in VLR, they are inherited from a common, abstract base type, Generic Location Register (GLR). Besides the node types, we have to define the relations as well: GMSC - GLR, GLR - MSC, MSC - BSC, BSC - BTS.

Using the metamodel, we can generate weaving contexts similarly to graphical programming languages. The concrete syntax definition is shown in Fig. 7.13/b, while Fig. 7.13/c shows an example model using the visualization.

Filter Protocol DSL

We have created a solution to model mobile communication visually [Mészáros, 2007] [Mészáros et al., 2007b]. The solution contains three languages: the first describes the devices participating in the communication, the second defines a protocol stack to define the communication protocol, while the aim of the third language – the Filter Protocol language – is to visualize one level of the protocol stack. VMTS Presentation DSL was used only in modeling the third level, because this language was the only one, where the appearance was important.
Filter Protocol is a special kind of data flow diagram. A Filter Protocol model can describe, for example, a cryptographic algorithm, where input data is encoded or decoded. Filter Protocol models can be built upon each other in a stack, where upper levels receive data directly by lower levels. The undermost filter receives its data directly from network, while the uppermost filter passes its output to a network application.

Model items in the filter are steps of data processing. Each filter has a tuple, a set of data, which has the same structure in all steps (in all model items) of the model. Model items of filters are required to receive and send data only in this data structure, but they can access part of the tuples inside. The language (Filter Protocol Metamodel) contains eight model items and a universal relation type.

Element is an abstract type which is mainly used to express that the relation can connect model items of any type. This relation is directed, thus, it defines the direction of data processing. MetaStart and MetaStop represents the start and the end of the data flow described by the Filter Protocol model. MetaVariable is used to create or allocate memory for parts of the tuple, or for the whole tuple of the filter. MetaPersistentStore can store data required by a later operation. The MetaPost element is the connection between different filters. A lower level filter can send data to an upper level using a Post item. The MetaOperation item represents a generic operation. There are built-in operations such as string or arithmetic operations, but the user can customize these operations as well using the Imperative OCL language. MetaDecision element can have more, than one outgoing edge, it can choose from these edges according to the result of the condition defined inside the decision item.
The concrete syntax of Filter Protocol contains weaving nodes for each model item shown in the metamodel (Fig. 7.14/a) except for Element, which is an abstract base model item, thus, it is not displayed in the subject models. An example model is shown in Fig. 7.14/b.

The ARIS language family

ARIS is a market-leading technology for Business Process Management [ARIS, Official Homepage, 2007]. ARIS is a set of modeling languages and modeling tools. It is used in several huge companies to rule methodology conventions in certain fields. A subset of ARIS languages has been used in one of the biggest mobile company of Hungary. To show the practical relevance of VMTS Presentation DSL we have created a tool to support the selected modeling languages of ARIS. More precisely, we have modeled the language rules by metamodels and the visualization by VMTS Presentation DSL.

Creating these definitions have taken approximately one week, while specialized modeling tools supporting almost the same languages have been developed for months. 'Almost the same' means in this case that the languages created by us have used minor simplifications, which were also acceptable by the company. Note that this large difference in development time in creating a modeling tool for a well-defined task is taken from a real world example, from a concrete, existing industrial project. This proves that our approach is effective and efficient for creating visual languages.

7.2.4 Constraint Optimization

To show the practical relevance of constraint optimization algorithms, a case study is provided. The case study describes the computers of a render farm (a computer cluster to render computer generated imagery). The metamodel of the case study is provided in Fig. 7.15/a.

The members of the cluster are modeled by NetworkNodes that can communicate with a local computer and define the type of network connection leading to the node as well. Computers are single desktop computers, with possibly more than one VideoCard (using the SLI technology). Moreover, Computers can have a special graphical CoProcessor, which can simulate certain graphical functions by software emulation. Fig. 7.15/b shows a concrete example instantiation of the metamodel.
In a concrete case, we use the modeled system to compute an animation based-on pixel shader calculations in parallel. To achieve this, we need a constraint that specifies that the network connection is fast enough and each computer can either emulate pixel shaders via its coprocessor (and has enough memory for calculations), or all video cards of the computer supports this function and has enough memory to compute a piece (e.g. a keyframe) of the animation. These requirements are described by the following constraint:

```plaintext
context NetworkNode::PixelShaderSupport : Boolean
    self.NetworkSpeed>=100 and
    (self.computer.Memory>512 and
     self.computer.coprocessor.PixelShaderSupported)
    or
     self.computer.videocard->forall(Card |
         Card.PixelShaderSupported and Card.Memory > 64 )
```

This original version of the constraint uses 24 model queries: (i) two queries to obtain the speed of the network, (ii) three queries to obtain the size of memory in the computer, (iii) four queries to check whether the coprocessor supports pixel shade emulation, (iv) three queries to get the list of connected video cards and (v) additional four queries for each video card to obtain their attributes.

If the RelocateConstraint algorithm is used as optimization, then the constraint is relocated to context Computer, thus, the number of queries is reduced to $3+2+3+2+4 \times 3 = 22$. Note that obtaining the network speed requires 3 queries because we have to navigate back to NetworkNode first.

By applying the ReferenceCaching algorithm we can reduce the number of model queries even more. The self reference is cached, thus, we save 3 queries. Moreover, inside the forall loop we can cache the video card queries and save another 4 queries. As result, instead of the original 24 queries, we need only 15, which is 62.5% of the original number. We have found...
that in general, real life examples this ratio is not that promising, but still, the optimization can accelerate the validation process by approximately 10-15%.

7.2.5 Other Applications

VMTS Presentation Framework (VPF) have been used not only in the aforementioned projects, but in several other industrial projects as well.

- In VMTS, model transformations are described by VMTS Control Flow Language, which is a visual language. Similarly, rewriting rules are also represented by a special visual language. Moreover, one of our most important current researches aims at creating a visual language to support QVT transformations as well. VPF provides an easy-to-use environment to construct these visual languages, and it plays an essential role in describing model transformations visually.

- Graphical user interface editors for mobile phones have been created by using VPF [Forstner et al., 2006]. Supported mobile platforms are the Windows Mobile [.NET Compact Framework, 2003] [Windows Mobile Smartphone, 2007] and the Symbian [Ortiz, 2003] platforms. In most of the related projects, these visual model definitions are transformed to source code that implements a mobile application. By customizing the transformations, it is able to unify different model platforms because of the high abstraction level of resource models [Lengyel et al., 2006g]. Without VPF, it would be hard to create and handle these languages, these models.

- Generative Programming (GP) focuses on modeling a whole software family system rather than a simple system. In order to support GP, we have created the Feature Modeling Language, a special domain-specific visual language that supports modeling the features of a system. The language has been developed using VPF. At the moment, our feature modeling solution is used by an international manufactory to model supported/unsupported features in device families.

7.3 Chapter Summary

In this chapter, we have presented the practical results, which are based on the theoretical constructs, algorithms and solutions presented earlier in the thesis. The chapter has introduced the Visual Modeling and Transformation System (VMTS) that was used as an implementation basis. The chapter has focused on the the visual language definition and transformation capabilities of VMTS.

The underlying modeling structure of VMTS is a labeled directed graph. Using the underlying graphs, VMTS defines an n-layer metamodeling system. The system consist of nodes, edges, and association nodes in a topological manner, while the attributes of the model items are defined as labels, which follow the n-layer attribute algebra presented in Section 4. The component AGSI offers a high-level interface to reach this graph structure. VMTS support an
Chapter 7. Application of the Results

RDBMS-based and a memory-based model factory as well. The model factories can be used with the same AGSI interface.

VMTS Presentation Framework (VPF) provides a flexible, customizable, plug-in-based architecture to simplify creating and editing visual languages. VPF support n-layer attributes and several other, promising features. Using the plug-ins, it is simple to define an editor for an arbitrary visual language. Moreover, to enhance the language definitions process, there is a plugin, VMTS Presentation DSL, which allows defining the appearance, the visual definition of the target language. VMTS Presentation DSL is based on the specialized concrete syntax metamodel presented in Section 4.3.

VMTS contains an OCL compiler component to support defining precise visual models. The compiler has built-in support for n-layer metamodeling constructs and it applies optimization algorithms automatically to reduce the number of model queries. This flexible, optimizing compiler is an efficient method to improve the precision of modeling languages.

VMTS supports not only the efficient creation of visual languages, but their transformation and conversion as well. VMTS Control Flow Language is a visual language that allows defining model transformation on a high-level of abstraction. The transformation rules are specified by using another visual language and Imperative OCL to precisely define the application steps of the selected rule. VMTS Parallel Transformation Engine enhances the performance of model transformation by applying the transformations in parallel.

Four case studies have been presented to prove the industrial applicability of the presented methods. The first case study shows the importance and flexibility of VMTS Presentation Framework and the attribute algebra presented in Chapter 4. The second and third case studies show how the VMTS Presentation DSL (Chapter 4) has been used in industrial projects. The fourth case study has elaborated a concrete model and constraint definition to show how the OCL optimization algorithms of Chapter 5 work. The case study has shown the practical relevance of optimization by performance measurements. Finally, several other industrial case studies have been introduced that use the contributed theoretical basis.
Chapter 8

Conclusions

8.1 Summary

The results that I have provided in this work are summarized in three theses. I have proven these results with engineering and mathematical methods, and I have illustrated their practical relevance in engineering applications. In this chapter, I give an overview on the theses and I outline some future directions of research and applications.

Thesis I

Related publications: [Mezei et al., 2006n] [Mezei, 2006a] [Levendovszky et al., 2005] [Mezei et al., 2005f] [Mezei et al., 2005g] [Lengyel et al., 2005c] [Mezei et al., 2005c] [Mezei et al., 2005d] [Mezei et al., 2005b] [Mezei et al., 2006m] [Mezei et al., 2006e] [Mezei, 2006b] [Mezei et al., 2006g] [Mezei et al., 2007e].

To support the definition of visual languages I have provided an attribute structure, and an attribute instantiation transformation that is applicable in n-layer metamodeling. Moreover, I have given a method to model the appearance definition of visual languages and a method to map the appearance definition to the topological definition provided by the metamodel.

- I have introduced an attribute structure referred to as Simplified N-layer Attribute Algebra. I have formalized the structure and I have given a formal definition for valid attribute configurations (valid models w.r.t. the attribute configuration). I have defined a layer-transparent instantiation transformation for the attribute algebra.

- I have improved (extended) the Simplified N-layer Attribute Algebra to support deep instantiated attributes, the new algebra is referred to as Enhanced N-Layer Attribute Algebra. I have formalized the attribute structure and a new instantiation transformation supporting the enhanced algebra.

- I have proven that applying the instantiation transformation on models following the Enhanced N-Layer Attribute Algebra always terminates.
Chapter 8. Conclusions

- I have introduced *semi-free instantiations* of models, which means instantiating the model and modifying the result model by not breaking the validity rules. I have proven that the validity of the original model grants that the n-ary self-embedding of the semi-free instantiation function exists and it produces valid models.

- I have shown that it is possible to construct a visual language that can model the appearance of other visual languages. I have defined this language using metamodeling.

- I have provided a method – based on model transformation – that can map and bind the appearance definition to the topological definition of the selected visual language and I have proven that this process can be automated, namely, the same method can be used regardless of the selection of the visual language.

- I have proven that the model transformation that binds the appearance and topological information always terminates.

Thesis II

Related publications: [Mezei et al., 2006a] [Mezei et al., 2007f] [Mezei et al., 2006j] [Mezei et al., 2006c] [Mezei et al., 2006h] [Mezei et al., 2006f] [Mezei et al., 2006i] [Mezei et al., 2007d] [Mezei et al., 2006l] [Mezei, 2006c] [Mezei et al., 2007b] [Mezei et al., 2006d] [Mezei et al., 2007g].

To support the efficient handling of precise and complete modeling languages, I have supplied an extension for OCL that supports the n-layer attribute algebra. I have also given two optimization algorithms to reduce the time of constraint evaluation.

- I have shown that OCL can be extended to support n-layer metamodeled attributes following the attribute algebra provided in Thesis I. I have provided an OCL dialect containing this extension.

- I have supplied OCLASM, a new formalism of OCL. I have shown that it is capable of describing the aforementioned OCL dialect and that it supports describing the dynamic behavior of constraints during evaluation.

- I have proposed an algorithm that can reduce the number of queries based on constraint relocation. The path between the original and the new context is referred to as *Relocation-Path*. I have shown that stepwise application of the relocation along this path does not restrict the relocation possibilities, thus, all valid relocation can be modeled by a sequence of relocation steps between direct neighbors.

- I have shown that a constraint must be reformulated in order to use it in the new context (after relocation). I have proven that the multiplicities of the underlying metamodel can affect this reformulation. I have elaborated all possible multiplicity combinations between two model items, and I have proposed constructs to handle the different cases. I have
proven that the relocation is not possible if the destination side allows zero multiplicity, but it is always possible in any other cases.

- I have given a formal definition of the relocation algorithm in OCLASM, and I have proven its correctness formally.

- I have supplied another optimization algorithm providing an efficient cache handling method. The algorithm makes possible to use local variables instead of model queries. I have formalized the algorithm and I have proven its correctness.

**Thesis III**

**Related publications: [Mezei et al., 2007c] [Mezei et al., 2006k] [Mezei, 2007] [Mezei et al., 2007a].**

I have introduced a truly parallel model transformation approach. I have supplied distributed algorithms, their synchronization and I have validated the performance gained from parallelization by measurements.

- I have defined the relation *metaconflict* between metamodel-based rules in a transformation. I have proven that those rules for which the relation is not satisfied are always applicable in parallel regardless of the host model.

- I have defined *independence blocks* in which all rules are applicable in parallel. The construction of these blocks is based on metaconflict examination. I have shown that the transformation-level parallelism is possible in this case. Transformation-level means that the rules are selected to be applied in parallel.

- I have given a set of algorithms that together allow executing independence blocks overlapped. Since in this case conflicts may occur between the rules, thus, I have supplied heuristic algorithms to avoid the conflicts, and an algorithm to resolve conflicts, if they still occur. I have proven that this modified algorithm can also be used in transformation-level parallelism.

- I have shown that in case of metamodel-based rewriting rules, the host model can be partitioned along the metamodel items of the rule, and the size of metamodel partitions affect the time of matching. I have given an exact, refined formula for the cost of matching:

\[
\prod_{i=1}^{k} \left( S(rn_i) - \sum_{j=1}^{i-1} N(rn_i, rn_j) \right),
\]

where \( k \) is the number of nodes in the selected rule, \( rn \) denotes the nodes in the rule, and \( N \) is a function defined as:

\[
N(l, m) = \begin{cases} 
1, & \text{if } \mathbb{T}(m) \in \mathbb{I}(\mathbb{T}(l)) \\
0, & \text{otherwise}
\end{cases}
\]
I have provided a rule-level parallel algorithm. Rule-level means that matching of a single rule is computed in parallel. The algorithm selects a (meta)node in the rule definition and creates a list of possible pairs (pivot points) in the host model. The elements of this list are computed in parallel. I have shown that the complexity of matching is

\[ k \prod_{i=2}^{k} \left( S(rn_i) - \sum_{j=2}^{i-1} N(rn_i, rn_j) \right) \times S(rn_1)/c \times h, \]  

(8.2)

where \( k \) is the number of nodes in the rule, \( rn_1 \) is the selected rule node, \( c \) is the number of clients and \( h \) is the quotient of calculating a match locally and sending/receiving a match via the network defined by its pivot point.

I have given another rule-level parallel algorithm that randomizes the order of evaluation for the pivot points (not restricted to the first selected rule node). I have shown that all clients evaluate all possible pivot points, but in possibly different order.

I have unified the transformation-level and rule-level parallelism algorithms and constructed the truly parallel transformation approach in a theoretical manner.

I have analyzed the performance of parallel execution by a concrete case study and measurements. According to the results, I have shown that the main weakness of transformation-level parallelism is that it cannot apply those transformations efficiently in parallel, which contain rules requiring considerably different times to match. However, this issue is solved by rule-level parallelism.

8.2 Application of the Theoretical Results

Related publications: [Lengyel et al., 2006a] [Lengyel et al., 2006e] [Lengyel et al., 2005d] [Lengyel et al., 2005b] [Lengyel et al., 2006g] [Mezei et al., 2005e] [Lengyel et al., 2007a] [Lengyel et al., 2006h] [Forstner et al., 2006] [Lengyel et al., 2006i] [Lengyel et al., 2006c] [Lengyel et al., 2006f] [Lengyel et al., 2006d] [Mezei et al., 2006h] [Lengyel et al., 2006b] [Mészáros et al., 2006] [Mezei, 2006a] [Mezei et al., 2005c] [Lengyel et al., 2005c] [Lengyel et al., 2005b] [Mészáros et al., 2007a] [Lengyel et al., 2007b].

The new scientific results are implemented in the Visual Modeling and Transformation System (VMTS). The following statements have been shown by the application:

- The Simplified and the Enhanced N-layer Attribute Algebra and their instantiation transformation can be realized. I have created an XML-based and an object-oriented representation of the algebra as reference implementations as well. The attribute algebra is capable of describing the attribute structure of several technological problems including,
but not limited to UML diagrams, resource editors for mobile phones and VMTS Control Flow Language.

- The appearance of visual languages can be described by a special domain-specific language. The method is capable of defining the appearance of Nessie-Schniedermann, Flowchart diagram, several network modeling DSLs, and ARIS.

- The containment hierarchy of model items can be separated to a real containment hierarchy and a logical hierarchy. The hierarchies can be synchronized.

- An OCL compiler can be constructed, which supports n-layer attributes and optimization as well.

- The memory-based model transformations are approximately three magnitude faster than the RDBMS-based transformations containing the same matching and rewriting algorithms.

- The truly parallel model transformation can be realized by using a double layered server-client architecture.

The following components of VMTS have been implemented based on the theoretical results presented in the thesis:

- I have developed the VMTS Presentation Framework (VPF) that is a metamodeling framework for editing n-layer metamodels. VPF offers a user-friendly modeling environment that is flexible in language creation and using the language models as well. I have shown that the plugin based architecture of VPF allows defining and customizing languages for various domains such as resource editors for mobile phones, model transformation languages, UML diagrams, or feature modeling diagrams.

- I have created the Attribute panel component that can force the metamodeling rules (the attribute structure defined by the metamodel) automatically. I have created a plugin and a domain-specific language, the VMTS Presentation DSL that can define the appearance of model items.

- I have created the OCL Compiler component for VMTS. I have proven that the compiler can be used in evaluating metamodeling constraints. I have shown that the OCL Compiler can be used in defining validated model transformations defined in [Lengyel, 2006].

- I have created the AGSI Compact Framework component that realizes a memory-based model factory. I have used this component to create AGSI Parallel Transformation Engine, which applies transformations in parallel.
Chapter 8. Conclusions

8.3 Future Work

Future work includes several directions. This section summarizes the main areas of future research.

- The Enhanced N-Layer Attribute Algebra is flexible enough to describe the attribute structure in almost every case. However, it would be useful to extend the attributes with operations. It is hard to find a solution that can describe these operations in a programming language independent way. Imperative OCL seems an ideal language for this purpose, but further research is needed to examine whether it is as flexible as the attributes are, in other words, it must be examined, whether ImperativeOCL can describe all possible operations needed in n-layer metamodeling.

- It would be beneficial to create a converter transformation that can convert OMG-based attributes to the attributes of the Enhanced N-Layer Attribute Algebra to support the reverse direction.

- Currently, the VMTS Presentation DSL supports only the appearance definition of visual languages, but not their behavior. It would be beneficial to have a method that defines exactly which editor operations are allowed in model items (for example a model can be docked into another model item, if a certain condition is true).

- Formalizing other OCL dialects by using OCLASM. Supporting other constraint types, than invariants. Analyzing properties of the constraints according to their formalization.

- Theoretically, we could use the maximum number of references in constructing the cache as well. This would mean that those expressions are cached, which are possibly referenced more than once. This would increase the memory required during the evaluation, but increase the scope of cached attributes. Note that this would not mean large additional overhead, because the method used in the calculation is almost the same. Future work includes to support selecting minimum or maximum reference mode as an option in the compiler.

- Developing new constraint optimization methods.

- Identifying new factors that have effect on the parallelization possibilities. Applying more complete performance tests.

- Creating a transformation-level parallelism algorithm that support amalgamated rules and distributed graphs. Comparing the performance of the approaches.

- Creating a formula that expresses the number of steps needed by the rule-level algorithm based on pseudo-random path.

- Inventing new transformation-level and rule-level algorithms based on practical and industrial feedback.
The field of visual language definition and transformation and the model transformation-driven model-based development are a constantly evolving area. The results of this thesis are recent results of a relatively new discipline, and hopefully, they will be a useful part of the visual language-based software development.
Appendix A

Detailed Proofs

Proof for Proposition 4.9

Proof. The instantiation rule is applied on model items one-by-one, thus, the cost of the instantiation can be calculated as the summary of instantiating each model item separately.

Firstly, we examine how much model query is required to process a single attribute. If the attribute under examination is an Attribute (line 7), then the algorithm creates the minimum number of instantiations allowed (line 9). Moreover, if the attribute have a complex type definition, then the algorithm calls itself recursively in order to process the components of the complex type. Inside the recursion, it is possible that one or more of the components are of complex types, because nested complex types are allowed. However, the depth of recursion is limited by the depth of the model item. In a single recursion step, we have to process at most $c_i$ components of a complex type and create $a_i$ attributes from each of these components. This means that $O(a_i \cdot c_i)$ model queries are used in a single recursion step and we create $O(a_i \cdot c_i)$ semi-processed subattributes. A subattribute in this case means a component of a complex attribute. By continuing the recursion, we require the same amount of model queries to process each subattribute. On the whole, we need $O((a_i \cdot c_i)^2)$ model queries on the first two recursion level and $O((a_i \cdot c_i)^n)$ on the $n$ level. Since the depth of the model item is $depth_i$, thus, processing an Attribute of the model items required at most $O((a_i \cdot c_i)^{depth_i})$ model queries.

Handling InstanceName (Line 2-5) needs a constant number of model queries, thus, it does not affect the complexity. Processing the model item – if there are only Attributes defined – requires $O(n_i \cdot (a_i \cdot c_i)^{depth_i})$ model queries.

If the attribute under examination is marked to be re-instantiated, the code segments from Line 17 - 22 are used. These steps map the exact structure of the selected metamodel attribute to the instance model item. A re-instantiated attribute can have at most $c_i$ children, which are processed by the recursive call in Line 22. If a child of a re-instantiated attribute is neither re-instantiated, nor is it an Attribute, then it cannot generate further recursion steps, because neither the condition in Line 7, nor the condition at Line 16 is satisfied. Nevertheless, if the child of a re-instantiated attribute is also marked as re-instantiated, then it is included in $r_i$. This means that we need $O(r_i \cdot c_i)$ steps in the worst case to process these attributes. Moreover, if we allow defining Attributes inside re-instantiated attributes, then $O(r_i \cdot c_i \cdot (a_i \cdot c_i)^{depth_i})$
steps can be required, because each children of each re-instantiated attribute can define a new \textit{Attribute}.

By summarizing the results, we obtain that the computational complexity for a single model item is

\[
\mathcal{O}(n_i \ast (a_i \ast c_i)^{depth_i}) + (r_i \ast c_i \ast (a_i \ast c_i)^{depth_i})).
\] (A.1)

Thus, if we instantiate a model with \(m\) model items, the transformation requires

\[
\mathcal{O}(\sum_{i=1}^{m} (n_i + r_i \ast c_i) \ast (a_i \ast c_i)^{depth_i}).
\] (A.2)

steps, which is the complexity of the original statement in the proposition.

\textbf{Proof for Proposition 4.11}

\textit{Proof.} The proof is based on induction. If \(n = 0\) then \(M_N = M_0\), which is a valid model by definition. We suppose that the induction condition is true for \(k\) (\(M_k\) is a valid model) and we show that it is also true to \(k + 1\). Since \(M_k\) is a valid model, thus invalidity can be caused only by instantiating \(M_k\), or by the semi-free changes made on the instantiation later.

The instantiation rule creates a valid model from \(M_k\), because

- \(\varphi_{\text{InstName}}\) is satisfied by Line 4 of Alg. 6.1.

- We do not add \textit{Attributes} during the instantiation, because they are not added automatically, and \(\varphi_{\text{ReservedNames}}(M_k)\) does not allow creating an attribute with the name 'Attribute'. Therefore, \(\varphi_{\text{Attr}}\) is satisfied. For the same reasons, \(\varphi_{\text{AttrName}}\) is satisfied as well.

- Similarly to 'Attribute', we cannot define at attribute with the name 'ComplexType' because of \(\varphi_{\text{ReservedNames}}(M_k)\), thus \(\varphi_{\text{CplxType}}\) and \(\varphi_{\text{TypeName}}\) are always satisfied.

- The formula \(\varphi_{\text{ReservedNames}}\) states that we cannot create an \textit{Attribute}, or \textit{ComplexType} that uses the reserved names. This is always satisfied, because the instantiation does not (cannot) create either of these attribute types.

- The first part \(\text{Name}(ID) = '\text{InstanceName}' \rightarrow (\neg \text{ReInstantiate}(ID))\) formula \(\varphi_{\text{ReInst}}\) is satisfied by Line 5. Other parts are satisfied for the same reasons as in the previous case.

Therefore, using the instantiation rule on \(M_k\) we obtain a valid model. Nevertheless, according to the definition of semi-free instantiation, validity cannot be broken by later, semi-free changes, thus \(M_{k+1}\) is a valid model. Thus, the statement of the proposition is always true.

\textbf{Proof for Proposition 4.12}

\textit{Proof.} At first, the transformation rules are examined whether they can affect the termination properties. The initial, final step, and the \textit{CreateNamespace} and \textit{ClearHelperInformation}
rules are executed only once. They are not exhaustive, thus, they do not affect the termination. In contrast, the loop containing the \textit{GetUnprocessedNode}, the decision object and the \textit{MatchAppearances} step are critical. When the transformation runs, the loop is executed until \textit{GetUnprocessedNode} can be matched. We unify the execution of consequent rules in the loop, namely, we create the E-based composition of the rules, a new rule that has an equivalent effect on the host graph. The key of the proving method is to show that this unification produces an LHS sequence that exceeds all limits.

The first step in the E-based composition is to unify a single execution of \textit{GetUnprocessedNode} and \textit{MatchAppearances}. Fig. A.1/a shows the composition in detail. No other composition structure is valid, because of the external causality between the rules. Empty and crossed circles represent weaving contexts, the cross in the circle means that the \textit{IsProcessed} attribute is set to \textit{true}. Filled circles are used to show appearance definitions. The generated model is not shown, because it is just an output model, nodes in the output model are never matched in the rules of the transformation.

![Fig. A.1. E-based composition](image)

Next, the composition is further composed by the next step in the loop. In this case the first rule is the composite rule, the another rule is \textit{GetUnprocessedNode}. The composition step is shown in Fig. A.1/b. The newly selected weaving context has an additional circle to show that it is different from the original one. It can be seen that \( R_{21} \) and \( L_{22} \) cannot be the same node, because \( R_{21} \) has the \textit{IsProcessed} attribute set to \textit{true}, thus, it cannot be matched again. This means that the composition represented by the figure is the only possible cumulative infinite LHS sequence. Thus, it is enough to examine whether the size of this sequence increments without any limit.

Every time the loop is executed, at least one new node appears in the LHS of the composed rule. Thus, for the LHS sequence in the E-based composition, it holds that

\[
\lim_{i \to \infty} |L^*_i| = \infty. \tag{A.3}
\]

This means, the transformation always terminates according to the Theorem 1.
Chapter A. Detailed Proofs

Proof for Proposition 5.19

Proof. The algorithm applied by the RelocateConstraint rule is correct, if none of the constructs (modification of the constraint) modifies the result of validation, i.e. if the set of valid models remains the same. Therefore, the presented proof shows that the steps of the rule are correct by mapping expressions of the original constraint to expressions in the new constraint and show that they are equivalent. The proof uses the CheckModelInvariants rule from Section 5.2.2. For sake of simplicity, the condition expression at line #7 in the algorithm is referred to as ValidModelCondition. We refer to sections of the rules as Rule(#From-To), for example, RelocateConstraint(#2-5).

Note that the aim of the proof method is to show the equivalence of the original and the relocated constraint, therefore, it is not stressed whether the rule RelocateConstraint is optimal.

To distinguish the original and the relocated constraints we use the labels "old" and "new", for example $C_{old}$ (that is the original constraint), or self old (self reference of the original constraint). Furthermore, for the sake of simplicity we suppose that the constraint was originally located in type $A$, while the new location is in type $B$. The instantiations of type $A$ are $a_1, a_2 \ldots a_n$ (collected in a set $Inst_A$), the instantiation of type $B$ are $b_1, b_2 \ldots b_n$ (collected in the set $Inst_B$). $B$ can be reached from $A$ using the name Dest $(To(a_i, Dest) = b_j)$ while the reverse direction uses Src $(To(b_i, Dest) = a_j)$. If the model allows zero multiplicity, then one of the following formulas are true:

$$\varphi_{ZeroOnSrc} : Mul(A, Dest)[0] = 0 \quad (A.4)$$

$$\varphi_{ZeroOnDest} : Mul(A, Dest)[2] = 0 \quad (A.5)$$

Firstly, we prove that the algorithm is always correct in these cases, then we formalize other possible cases as follows:

$$\varphi_{ExactlyOne} : Mul(A, Dest)[1] = 1 \land Mul(A, Dest)[3] = 1 \quad (A.6)$$

$$\varphi_{ManyOnSrc} : Mul(A, Dest)[1] > 1 \land Mul(A, Dest)[3] = 1 \quad (A.7)$$

$$\varphi_{ManyOnDest} : Mul(A, Dest)[1] = 1 \land Mul(A, Dest)[3] > 1 \quad (A.8)$$

$$\varphi_{ManyBoth} : Mul(A, Dest)[1] > 1 \land Mul(A, Dest)[3] > 1 \quad (A.9)$$

If the constraint does not contain any model query, then the constraints results in a simple Boolean value true, or false. The result does not depend on the underlying model, thus it is always true, or false.

If this constant result is true, then it means that all possible models are valid. This means that ValidModelCondition is always satisfied, thus, the inner forall expression at CheckModelInvariants(#3-9) can be replaced by a constant true value. Therefore $C_{new}$ always evaluates to true as well.

If this constant result of the constraint is false (no model is valid, which contains a model item of the selected type), then ValidModelConditionOld is not satisfied for instances of $A$,
while ValidModelCondition\textsubscript{New} is not satisfied for instances of $B$. Therefore, the relocation of the constraint is not correct if Inst\textsubscript{A} or Inst\textsubscript{B} is empty.

Inst\textsubscript{B} can be empty only if formula $\varphi_{\text{ZeroOnDest}}$ holds (zero multiplicity is allowed on the destination side). This case is handled at RelocateConstraint(#2-3): the condition checks whether for each $a_i$ there exists at least one $b_j$, more generally that there is a $b_j$ in the model, which can be used as a host for the constraint (host nodes of a constraint are nodes, in which the constraint is defined). If not, the rule throws an error message.

Similarly, Inst\textsubscript{A} can be empty if formula $\varphi_{\text{ZeroOnSrc}}$ holds. The problem is solved by the condition at RelocateConstraint(#4-9), which ensures that the constraint is evaluated only to those $b_j$s which are connected to at least one $a_i$ (otherwise it returns with a constant true). Thus, if Inst\textsubscript{A} is empty, then the constraint is not evaluated.

As result of the conditions at RelocateConstraint(#2-9), the following formulas always hold:

\[
\varphi_{\text{AToB}}(i) = \forall i : \text{Meta}(i) = A \rightarrow (\exists j, \exists \text{Dest} : \text{Meta}(j) = B \land j \in \text{To}(i, \text{Dest}) \quad (A.10)
\]

\[
\varphi_{\text{BToA}}(i) = \forall i : \text{Meta}(i) = B \rightarrow (\exists j, \exists \text{Src} : \text{Meta}(j) = A \land j \in \text{To}(i, \text{Src}) \quad (A.11)
\]

Note that these formulas ensure also that the relocation of constraints without model queries is always correct.

If the constraint contains model queries, then the result of eval at CheckModelInvariants(#8) can be affected by the navigations and attributes of $a_i$. The relocation is correct if the result of model queries in the original context is the same as the result in the new context. Different cases are indexed by the formulas defined above:

\begin{itemize}
  \item $\varphi_{\text{ExactlyOne}}$: RelocateConstraint(#24) and RelocateConstraint(#26) are used. RelocateConstraint replaces navigations from $A$ to $B$ with a self reference. Instead of self\textsubscript{Old}.Dest the new constraint has a self\textsubscript{New} expression (RelocateConstraint(#26)). According to the rules of OCLASM, the value of self\textsubscript{Old}.Dest is retrieved by the monitored function call To($a_i,"\text{Dest}"$), which results in $b_j$, an instance of $B$. Since the formula $\varphi_{\text{AtoB}}$ holds, thus, this $b_j$ always exists, therefore the expression To($a_i,"\text{Dest}"$) can be replaced by the value $b_j$ for this certain navigation expression for this certain $a_i$. However, the constraint is checked against all $a_i$s (CheckModelInvariants(#4)), thus To($a_i,"\text{Dest}"$) is always replaceable by an appropriate $b_j$. In the relocated constraint this replacement is done by using the new self reference. Note that relocation does not create false host nodes (nodes which could not affect the result of the original constraint, but could affect the result of the relocated version) for the constraint (due to $\varphi_{\text{BtoA}}$).

  \item RelocateConstraint inserts a navigation expression from $B$ to $A$ (RelocateConstraint(#24)) before any navigations/attribute queries from $A$ to anywhere, but not $B$. The expression self\textsubscript{Old}.x (where $x$ is not "Dest") is replaced by self\textsubscript{New}.Src.x. The proof is similar to the previous case, but on the reverse direction. Since the formula $\varphi_{\text{BtoA}}$ holds, thus self\textsubscript{New}.Src which results in a To($b_j,"\text{Src}"$) can be replaced by the value $a_j$ for this certain navigation/attribute expression for this certain $b_j$. Moreover, the new constraint is checked against all occurrences of $b_j$ (CheckModelInvariants(#4)), thus To($b_j,"\text{Src}"$) can always be replaced by an appropriate $a_j$. Therefore, the value of self\textsubscript{Old} always equals the value of self\textsubscript{New}.Src. Note that relocation does not delete host nodes because of $\varphi_{\text{AToB}}$.
\end{itemize}
\( \varphi_{ManyOnDest} \): The section RelocateConstraint(#19-22) and RelocateConstraint(#26) are applied. Correctness of navigations of the form \( self_{Old}.x \), where \( x <> "Dest" \) is ensured by RelocateConstraint(#26) according to constructs used in \( ExactlyOne \). However, the result of navigations from \( A \) to \( B \) results in a set of \( b_j \)'s. The condition at RuleConstraint(#19) checks whether the result of the navigation expression is used in a \( forall \) construct, if so, then the original expression is replaced by the inner expression of \( forall \). Thus, \( self_{Old}.Dest \rightarrow forall(Exp_{Old}) \) is transformed into \( Exp_{New} \). In OCL, \( forall(Exp) \) is a set operation, which is true only if \( Exp \) is satisfied for each item in the set. Therefore, the original expression \( self_{Old}.Dest \rightarrow forall(Exp_{Old}) \) is true for a certain \( a_i \), if \( Exp \) is true for each \( b_j \in To(a_i, Dest) \). This means that the original expression in a certain \( a_i \) can be replaced by \( Exp_{New} \) in \( b_j \in To(a_i, Dest) \). The original constraint is evaluated for each \( a_i \), thus, the replacement is correct in each \( b_j \) connected to one of the \( a_i \)'s. Moreover, since \( \varphi_{BToA} \) holds, thus, \( b_j \)'s are always connected to an \( a_i \). This means that the relocation is always correct in this case.

If the condition at RelocateConstraint(#19) is not satisfied, then the replacement inserts a navigation back to the original constraint and the expression is evaluated there. The expression \( self_{Old}.Dest \) is replaced by \( self_{New}.Src.Dest \). Because of \( \varphi_{BToA} \) this replacement is always correct as shown constructs used in \( \varphi_{ExactlyOne} \).

\( \varphi_{ManyOnSrc} \): The sections RelocateConstraint(#10-15) and RelocateConstraint(#26) are applied. The correctness of navigations of the form \( self_{Old}.Dest \) is granted by RelocateConstraint(#26) according to the constructs used in \( \varphi_{ExactlyOne} \). However, navigations from \( B \) to \( A \) results in a set of \( a_i \)'s. The condition at RelocateConstraint(#10) checks whether such navigation is required (using the rule \( ForAllCheck \)). Backward navigation to the original constraint is required in the case of \( self_{Old}.x \) expressions (where \( x <> "Dest" \), or if \( Mul(A, Dest) \)[1] > 1 according to constructs used in \( \varphi_{ManyOnDest} \). Here, only the first case is possible, which is handled by adding a \( forall \) expression to the constraint during relocation (RelocateConstraint(#10-15)). The new \( forall \) expression encapsulates the whole constraint and it simulates the backward navigation using the iteration variable 'OrigSelf', where navigation expressions are replaced by variable calls in AddBackNavigation(#3-4). Therefore, the constraint '... \( self_{Old}.x ... \)' is transformed to '\( self_{New}.Src \rightarrow forall(OrigSelf ... OrigSel.f.x ...') \). For a certain expression and \( b_j \), the replacement is correct for \( a_i \)S connected to \( b_j \). Since \( \varphi_{AToB} \) holds, thus, each \( a_i \) is connected with at least one \( b_j \) and evaluation checks each \( b_j \) of the model, thus, the replacement is correct for all \( a_i \)S of the model. Note that the outermost \( forall \) expression ensures that different navigation/attribute calls of the constraint use the same \( a_i \) (since the value of \( OrigSelf \) does not change), thus, the attributes/navigations of \( a_i \)S can always be distinguished.

\( \varphi_{ManyOnBoth} \): RelocateConstraint(#10-15) and RelocateConstraint(#19-22) are used. Firstly, an encapsulating \( forall \) expression is added if there is a navigation/attribute call of form \( self_{Old}.x \) (where \( x <> "Dest" \)) or a \( self_{Old}.Dest \) expression without \( forall \). This construction is borrowed from \( \varphi_{ManyOnSrc} \). Secondly, the \( forall \) expressions of the original constraint are replaced according to the constructs used in \( \varphi_{ManyOnDest} \).

The replacement of the expression \( self_{Old}.x \ (x <> "Dest") \) in a certain \( a_i \) is correct for each \( b_j \in To(a_i, Dest) \) according to constructs used in \( \varphi_{ManyOnSrc} \). Moreover, the replacement of these type of expressions is also correct in general (because of \( \varphi_{AToB} \)). However, it is possible
that the expression is evaluated in a certain $a_i$ more then once (more precisely once for each element of $T_o(a_i, \text{Dest})$).

The replacement of the expression $self_{Old}Dest \rightarrow forall$ is correct according to constructs used in $\varphi_{\text{ManyOnDest}}$. Multiplicity 'MoreThanOne' on the source side does not affect this correctness, because the updated expressions use navigations only from the context of $B$, thus, it is not important how many $a_i$s are connected with the current $b_j$.

This is not the case with expressions of form $self_{Old}Dest \rightarrow Exp$, where $Exp$ is not a $forall$ expression. Here, navigation back to the original context is mandatory, thus, the original expression is transformed into $self_{New}Src.Dest \rightarrow Exp$. Relocation is correct in this case if the value of $self_{Old}$ can be replaced by $self_{New}$. However, the function $\text{ForAllCheck}$ returns true at the condition at RuleConstraint(#10), which means that the constraint is encapsulated by a new $forall$ as in $\varphi_{\text{ManyOnSrc}}$. This $forall$ expression ensures that for a certain $b_j$, the constraint is evaluated for all items of the set $self_{New}Src$ separately (for all $a_i$s connected with $b_j$). Moreover, $\varphi_{\text{BTtoA}}$ holds, which means that all $a_i$s are checked by the relocated constraint. This means that the relocation is correct in this case as well.

Since the possible multiplicity combinations have tested, we have proven that the $\text{RelocateConstraint}$ rule is correct in all cases.

Proof for Proposition 6.3

Proof. We examine two rules of an arbitrary transformation $t$. The rules are chosen arbitrarily except that we choose them not to be in metaconflict. We define a coloring function $C$ for model all items of $G^M$ (the metamodel of $G$). Then, we color the rules, the host graph and their metamodels using three colors. Finally, we show that there is a total, non-overlapping splitting of the original host graph along these colors and neither of the rules crosses the border of splitted parts.

Nodes and edges ($x$) are colored by $C$ as follows:

- If $x$ is in $S(L^1_M) \cup S(R^1_M)$, but not in $S(K^1_M)$, then the color of $x$, and all other model items of $I(x)$ is set to blue.

- If $x$ is in $S(L^2_M) \cup S(R^2_M)$, but not $S(K^2_M)$, then the color of $x$, and all other model items of $I(x)$ is set to red.

- If $x$ is neither blue, nor red, then it is colored to green.

The rules that we are examining are not in metaconflict. This means that there does not exist a model item that must be colored to blue and red at the same time. Moreover, the model items of the first rule ($r^1_M: L^1_M \leftarrow K^1_M \rightarrow R^1_M$) are all blue, or green, while the model items contained by the second rule ($r^2_M: L^2_M \leftarrow K^2_M \rightarrow R^2_M$) are red or green.

Using the colors defined in the metamodel, we can color the nodes and edges of rules of $G$ by preserving the coloring function: $\forall x : T(C(x)) = C(T(x))$. Unambiguous coloring of model items on the instance level is always applicable, because both the metamodel item and the color of an item are uniquely determined. We obtain a fully colored $G$. 


In general, the original host graph $G$ consists of blue, red and green nodes and edges. According to the definition of graph rewriting in the DPO approach, when executing a rewriting rule $r_1$, firstly we remove nodes and edges, which are included in $S(L_1)$, but not in $S(K_1)$. More precisely, $S(G) \backslash S(m(L_1)) \cup S(m(I(K_1)))$ is applied. However, in this step we can remove only blue nodes and edges from $G$, because of the definition of metaconflict and the unambiguity of the coloring function.

Secondly, new nodes and edges are added to $G$. The new model items are included in $S(R_1)$ but not in $S(K_1)$, thus, these new items are also colored to blue in order to preserve coloring for the instance level. As result, at the end of the first rewriting rule, all items that have changed (deleted, or created) by the rule must be blue.

Similarly, when applying the second rewriting rule, only red nodes and edges can be added, or removed. This means that the original host graph $G$ can be partitioned along the colors and the order of execution is irrelevant. \hfill \Box
Bibliography


