ONLINE VALIDATION OF VISUAL MODEL TRANSFORMATIONS

Ph.D. Thesis

László Lengyel

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

Dr. Tihamér Levendovszky, Ph.D.

Budapest, 2006
# 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 5
2.1 Metamodeling and Constraints ........................................... 6
  2.1.1 The Unified Modeling Language ................................... 6
  2.1.2 The Object Constraint Language .................................. 6
  2.1.3 Domain-Specific Modeling ................................... 8
2.2 Model-Driven Development ......................................... 8
  2.2.1 Model-Driven Architecture ................................... 8
  2.2.2 Model Integrated Computing ................................ 9
2.3 Graph Rewriting ............................................. 11
2.4 A Case Study ............................................... 11
2.5 Open Issues ................................................ 13
2.6 The Visual Modeling and Transformation System .................. 14
2.7 Chapter Summary ............................................ 14

Chapter 3
Graph Transformation Approaches 16
3.1 The DPO Approach ............................................ 17
3.2 The SPO Approach ............................................ 18

Chapter 4
Validated Model Transformation with Efficient Constraint Checking 20
4.1 Backgrounds and Related Work ................................... 20
# TABLE OF CONTENTS

| 4.1.1 | Design by Contract                                                                 | 21  |
| 4.2   | Constraint Validation                                                              | 21  |
| 4.2.1 | Validating Metamodel Constraints During Metamodel Instantiation                    | 22  |
| 4.2.2 | Validating Transformation Rule Constraints During Model Transformation              | 22  |
| 4.2.2.1 | Relation between Pre- and Postconditions and OCL Constraints                   | 23  |
| 4.2.2.2 | The Constraint Pair Concept                                                      | 24  |
| 4.2.2.3 | General Validation                                                               | 24  |
| 4.2.2.4 | General Preservation                                                             | 25  |
| 4.2.2.5 | General Guarantee                                                                | 27  |
| 4.2.2.6 | Validation, Preservation and Guarantee Algorithms                                | 28  |
| 4.2.3  | The Rule Constraint Validator Algorithm                                           | 30  |
| 4.2.4  | The Invariant Analysis Algorithm                                                  | 31  |
| 4.2.5  | The Persistent Analysis Algorithm                                                 | 33  |
| 4.2.6  | The Optimized RCV Algorithm                                                       | 34  |
| 4.3    | Example for Validated Model Transformation                                       | 35  |
| 4.4    | Chapter Summary                                                                  | 37  |

| 5.1    | Backgrounds and Related Work                                                      | 40  |
| 5.1.1  | Aspect-Oriented Software Development                                             | 40  |
| 5.1.1.1 | General AOSD Concepts                                                            | 41  |
| 5.1.1.2 | Aspect-Oriented Programming                                                       | 44  |
| 5.1.1.3 | Aspect-Oriented Modeling                                                          | 45  |
| 5.2    | The Constraint Management Problem                                                | 47  |
| 5.3    | Applying Aspect-Oriented Techniques for Constraint Management                    | 48  |
| 5.3.1  | Aspect-Oriented Constraints                                                       | 49  |
| 5.3.2  | A New Type of Aspect: Constraint Aspect                                           | 49  |
| 5.3.2.1 | Optimizing the Weaving Process with Normalization                                | 52  |
| 5.3.2.2 | Creating Constraint Aspect from OCL Constraint                                     | 59  |
| 5.4    | Weaver Algorithms                                                                | 60  |
| 5.4.1  | Constraint Weaving                                                               | 61  |
| 5.4.1.1 | The Global Constraint Weaver Algorithm                                           | 62  |
| 5.4.1.2 | The Constraint Aspect Weaver Algorithm                                           | 63  |
| 5.4.1.3 | Comparison of the Weaver Algorithms                                              | 64  |
| 5.5    | Overview of Different Aspect-Oriented Constraint Notations                       | 65  |
| 5.6    | Example for Aspect-Oriented Constraint Management                                | 66  |
| 5.7    | Chapter Summary                                                                  | 66  |

| 6.1    | Backgrounds and Related Work                                                      | 70  |
| 6.1.1  | Termination Criteria for Contextual Layered Graph Grammars                       | 72  |
| 6.1.2  | Termination Criteria for High-Level Replacement Units                             | 74  |
| 6.1.3  | Termination Criteria for DPO Transformations with Injective Matches              | 75  |
| 6.2    | VMTS Visual Control Flow Language                                                 | 77  |
| 6.2.1  | Sequencing Transformation Rules                                                  | 78  |

**Chapter 5**

**Aspect-Oriented Constraint Management**

| 5.1    | Backgrounds and Related Work                                                      | 40  |
| 5.1.1  | Aspect-Oriented Software Development                                             | 40  |
| 5.1.1.1 | General AOSD Concepts                                                            | 41  |
| 5.1.1.2 | Aspect-Oriented Programming                                                       | 44  |
| 5.1.1.3 | Aspect-Oriented Modeling                                                          | 45  |

**Chapter 6**

**Control Flow Related Properties of Model Transformations**

| 6.1    | Backgrounds and Related Work                                                      | 70  |
| 6.1.1  | Termination Criteria for Contextual Layered Graph Grammars                       | 72  |
| 6.1.2  | Termination Criteria for High-Level Replacement Units                             | 74  |
| 6.1.3  | Termination Criteria for DPO Transformations with Injective Matches              | 75  |
| 6.2    | VMTS Visual Control Flow Language                                                 | 77  |
| 6.2.1  | Sequencing Transformation Rules                                                  | 78  |
### TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2</td>
<td>Branching with OCL Constraints</td>
<td>79</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Hierarchical Rules</td>
<td>80</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Iteration, Tail Recursion and Parallel Executions of the Rules</td>
<td>80</td>
</tr>
<tr>
<td>6.3</td>
<td>Visual Control Flow Language Algorithms</td>
<td>81</td>
</tr>
<tr>
<td>6.4</td>
<td>Composing Metamodel-Based Model Transformation Rules</td>
<td>83</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Self-Composing Metamodel-Based Model Transformation Rules</td>
<td>85</td>
</tr>
<tr>
<td>6.4.2</td>
<td>The Transformation Rule Composing Algorithm</td>
<td>90</td>
</tr>
<tr>
<td>6.5</td>
<td>VCFL Termination Properties</td>
<td>92</td>
</tr>
<tr>
<td>6.5.1</td>
<td>VCFL Control Flows with Non-Exhaustive Transformation Rules</td>
<td>92</td>
</tr>
<tr>
<td>6.5.2</td>
<td>VCFL Control Flows with Exhaustive Transformation Rules</td>
<td>93</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Termination Properties of VCFL Loops</td>
<td>94</td>
</tr>
<tr>
<td>6.5.4</td>
<td>VCFL Termination Algorithm</td>
<td>95</td>
</tr>
<tr>
<td>6.6</td>
<td>Example Transformation and Its Termination Properties</td>
<td>96</td>
</tr>
<tr>
<td>6.7</td>
<td>Chapter Summary</td>
<td>99</td>
</tr>
<tr>
<td>7</td>
<td>Chapter 7: A Synopsis of Model Transformation Systems</td>
<td>101</td>
</tr>
<tr>
<td>7.1</td>
<td>Model Transformation Approaches</td>
<td>101</td>
</tr>
<tr>
<td>7.1.1</td>
<td>GREAT</td>
<td>101</td>
</tr>
<tr>
<td>7.1.2</td>
<td>PROGRES</td>
<td>102</td>
</tr>
<tr>
<td>7.1.3</td>
<td>VIATRA</td>
<td>103</td>
</tr>
<tr>
<td>7.1.4</td>
<td>AGG</td>
<td>103</td>
</tr>
<tr>
<td>7.1.5</td>
<td>AToM³</td>
<td>104</td>
</tr>
<tr>
<td>7.1.6</td>
<td>FUJABA</td>
<td>104</td>
</tr>
<tr>
<td>7.1.7</td>
<td>OMG QVT</td>
<td>104</td>
</tr>
<tr>
<td>7.1.8</td>
<td>Other Approaches</td>
<td>105</td>
</tr>
<tr>
<td>7.2</td>
<td>Comparison of Model Transformation Approaches</td>
<td>106</td>
</tr>
<tr>
<td>7.3</td>
<td>Chapter Summary</td>
<td>107</td>
</tr>
<tr>
<td>8</td>
<td>Chapter 8: Application of the Results</td>
<td>108</td>
</tr>
<tr>
<td>8.1</td>
<td>The Visual Modeling and Transformation System</td>
<td>108</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Visual Model Processing in VMTS</td>
<td>110</td>
</tr>
<tr>
<td>8.1.1.1</td>
<td>Specification of Transformation Rules</td>
<td>110</td>
</tr>
<tr>
<td>8.1.1.2</td>
<td>VMTS Control Flow Support</td>
<td>112</td>
</tr>
<tr>
<td>8.1.1.3</td>
<td>Constraint Management</td>
<td>112</td>
</tr>
<tr>
<td>8.2</td>
<td>Model-Based Development with VMTS</td>
<td>114</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Model-Based Unification of Mobile Platforms</td>
<td>114</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Model-to-Model Transformation with Strictly Controlled Model Transfor-</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>mation</td>
<td></td>
</tr>
<tr>
<td>8.2.3</td>
<td>Model Transformation-Driven Software Maintenance</td>
<td>118</td>
</tr>
<tr>
<td>8.3</td>
<td>Chapter Summary</td>
<td>121</td>
</tr>
<tr>
<td>9</td>
<td>Chapter 9: Conclusions</td>
<td>123</td>
</tr>
<tr>
<td>9.1</td>
<td>Summary</td>
<td>123</td>
</tr>
<tr>
<td>9.2</td>
<td>Application of the Theoretical Results</td>
<td>127</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3 Future Work</td>
<td>128</td>
</tr>
<tr>
<td>Appendix A</td>
<td></td>
</tr>
<tr>
<td>Detailed Proofs</td>
<td>130</td>
</tr>
<tr>
<td>Appendix B</td>
<td></td>
</tr>
<tr>
<td>Computational Complexity Considerations on Constraint Evaluation and</td>
<td>140</td>
</tr>
<tr>
<td>Navigation</td>
<td></td>
</tr>
<tr>
<td>Appendix C</td>
<td></td>
</tr>
<tr>
<td>Comparison Table for Model Transformation Tools</td>
<td>144</td>
</tr>
<tr>
<td>Appendix D</td>
<td></td>
</tr>
<tr>
<td>Verification Tools</td>
<td>145</td>
</tr>
<tr>
<td>D.1 GROOVE</td>
<td>145</td>
</tr>
<tr>
<td>D.2 CheckVML</td>
<td>146</td>
</tr>
<tr>
<td>D.3 Angur</td>
<td>146</td>
</tr>
<tr>
<td>D.4 OBGG</td>
<td>147</td>
</tr>
<tr>
<td>Bibliography</td>
<td>162</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Model-Driven Architecture ................................. 9
2.2 The basic concepts of MIC [189] .......................... 10
2.3 Principles of VMTS metamodel-based model transformation .......... 15

3.1 An illustration of direct derivation: $L$, $K$, $R$ are the left-hand side, the interface and the right-hand side graph; $G$ and $H$ are the graphs before and after the rule firing and $D$ corresponds to $K$; $l$, $r$, $f$, $g$, $m$, $k$, $n$ are inclusions .......... 18
3.2 The direct derivation in the SPO approach .......................... 18

4.1 Constraint validation during metamodel instantiation ..................... 22
4.2 Constraint validation during model transformation .......................... 31
4.3 Transformation rule $CreateTable$ ............................. 36

5.1 (a) Mapping concerns $C_1, C_2 \ldots C_n$ to modules $M_1, M_2 \ldots M_n$, (b) Concern $C_2$ crosscuts modules $M_1$, $M_2$ and $M_{n-1}$ .......................... 42
5.2 (a) Crosscutting concerns ($C_2$ and $C_3$), (b) Join points, Tangling and Crosscutting concerns .......................... 43
5.3 Aspect separation .................................. 43
5.4 Traditional and AOP approaches .................................. 46
5.5 An overview of the AOM approach ................................. 46
5.6 (a) Example metamodel, (b-c) Two transformation rules built from metamodel types with crosscutting constraints .......................... 48
5.7 (a) Constraint Aspect, (b) Propagated Constraint Aspect, (c/1) Constraint Aspect, (c/2) Canonical Constraint Form, (c/3) Pure Canonical Constraint Form .......................... 51
5.8 A constraint aspect / left-hand side graph with propagated OCL constraints .......................... 52
5.9 (a) Example left-hand sides with multiplicity 0..* and, (b) Example matched input model .......................... 54
5.10 (a) Example left-hand side graph, (b) Normalized left-hand side graph, (c) Example matched host model .......................... 55
5.11 The example constraint aspect / LHS with normalized OCL constraints .......................... 57
5.12 Creating constraint aspect from OCL constraint and its normalization .......................... 60
5.13 The weaving process and the input and output of the GCW and CAW algorithms .......................... 61
5.14 An overview of the different aspect-oriented constraint notions and the presented algorithms .......................... 65
5.15 Transformation rule *ProcessAssociation* .......................... 67
5.16 The result of the constraint propagation with Global Constraint Weaver 67

6.1 VCFL model of the transformation *Class2RDBMS* .......................... 78
6.2 (a) The metamodel of the VMTS Visual Control Flow Language, (b) The metamodel of the VMTS Rule Editor .................................. 79
6.3 An example hierarchical rule ........................................... 80
6.4 The E-based composition of metamodel-based model transformation rules .... 84
6.5 Self-composition of *tree structure* transformation rules ............... 86
6.6 Multiplicity calculation during rule composition ................................ 88
6.7 Self-composition of *inserting structure* transformation rules ............ 89
6.8 Self-composition of *inserting* and *deleting structure* transformation rules ... 90
6.9 Example terminating creation rule ...................................... 93
6.10 Transformation rule *AddParentAssociation* ................................ 97
6.11 Transformation rule *ShiftParentClassHelper* ................................ 97
6.12 External causalities between rules *AddParentAssociation* and *ShiftParentClassHelper* ........................................... 98
6.13 The E-based composition of the composed metamodel-based model transformation rules: *ShiftParentClassHelper* $\ast_E$ *AddParentAssociation* ........................................... 98
6.14 External causalities between rules *ShiftParentClassHelper* and *AddParentAssociation* ........................................... 98

8.1 VMTS block diagram .................................................. 109
8.2 Transformation rule *CreateParentClassHelper* ............................ 110
8.3 Transformation rule and pattern rule node attributes .......................... 111
8.4 Block diagram of the case study ...................................... 115
8.5 Transformation rule (a) *Resource* $\rightarrow_{XML}$ and (b) *Statechart* $\rightarrow_{CodeDOM}$ 116
8.6 Example input class diagram ............................................ 117
8.7 (a) VMTS Relation Database metamodel, and (b) Generated database model 117
8.8 The input and output metamodels of the model evolution case study .......... 119
8.9 (a) The transformation rule of the case study, (b) Example input model and generated output model ...................................... 119
8.10 Example input model evolution ........................................ 120
8.11 Updated input and modified output models of the case study ............... 121

9.1 Future work: AOM + Weaving + Code generation .......................... 128

A.1 Self-composition of *tree structure* transformation rules .................. 134
A.2 Self-composition of *inserting structure* transformation rules ............. 136
A.3 Self-composition of *deleting structure* transformation rules ............. 138

B.1 (a) A navigation path, (b) Example metamodel with constraints, (c) Example model .................................................. 140
List of Tables

4.1 Truth table of the validation, preservation and guarantee properties ............. 24
4.2 The principles of the constraint pair creation (where C denotes a constraint) .... 24
4.3 Measurement results - The computational complexity of the constraint evaluation
during the matching process .................................................. 34

6.1 Semantics of VCFL elements .................................................. 82
6.2 Rules for calculating multiplicities during transformation rule composition .... 87

B.1 The calculation table of the navigations contained by Const1 (Fig. B.1b and Fig.
B.1c) ......................................................................................... 142
B.2 The calculation table of the navigations contained by Const2 (Fig. B.1b and Fig.
B.1c) ......................................................................................... 142

C.1 Comparison table of control flow, constraint and attribute transformation support
for model transformation tools ..................................................... 144
Abstract

Model-based approaches in development are widely recognized as a potential way of increasing productivity in software engineering. Model-based development is driven by model transformations that attempt to bridge the large semantic gaps between high-level models and low-level languages. There is a demand for researching the ways how model transformation can become more flexible, efficient, and highly-configurable as well as validated. This thesis addresses issues of visually defined validated model transformations. The results that facilitate the realization of such a system can be divided into three main parts.

The first contribution deals with the relationship between the pre- and postconditions and the Object Constraint Language (OCL) constraints enlisted in model transformation rules. This construct provides the background of an online validated transformation approach that involves the examination of individual transformation rules and the whole transformations as well. Furthermore, optimization algorithms are provided that facilitate the early detection of constraint failure, which makes the transformation process more efficient.

Validation in the transformation introduces a new concern that often crosscuts the transformation rules. The second result group provides an aspect-oriented constraint management approach to solve the problem of the crosscutting constraints. This method facilitates consistent constraint management in model transformation rules, furthermore, it makes the constraints as well as the transformation rules reusable. In addition, to accelerate the constraint evaluation during the transformation process, normalization algorithms are proposed to eliminate navigation steps from constraints.

The third contribution includes the realization a visual control flow language (VCFL) that facilitates sequencing transformation rules, branching with constraint-driven methods, hierarchical transformation rules, parallel rule execution along with iteration and recursion. Algorithms are provided to compose transformation rules. These algorithms support the prediction of the transformation behavior such as the termination of the transformations.

In order to illustrate the practical applications of the results, the Rule Editor, Control Flow Designer, Constraint Aspect Editor and Aspect-Oriented Constraint Manager components of the Visual Modeling and Transformation System have been developed. Its application includes model compilers for UML diagrams and other domain-specific languages, such as resource models. These methods facilitate the model-based development-driven unification of mobile platforms. Besides the model-to-code applications, the model-to-model transformations are also supported - for example generating relational database management system model from class diagrams. Furthermore, the approach can be applied to solve issues with respect to software evolution.
Összefoglaló

A szoftverfejlesztés hatékonyságát növelő módszerek közül a legtöbb figyelem a modell alapú fejlesztési megközelítésekre irányul. A modell alapú szoftverfejlesztés alapját a modelltranszformációk képezik, melyek betöltik a magas szintű modellek és az alacsony szintű nyelvek között kialakult szemantikus rést. A modelltranszformációkkal kapcsolatos kutatások célkitűzése a rugalmasabb, hatékonyabb, konfigurálhatóbb és nem utolsó sorban validált transzformációk elérése. Jelen értekezés témája a vizuális modelltranszformációk futásidejű validálása. Az eredmények, amelyek lehetővé teszik egy ilyen transzformációs rendszer megvalósítását, három fő részre oszthatók.

Az eredmények első csoportja az elő- és utófeltételek, valamint a transzformációs szabályokhoz rendelt Object Constraint Language (OCL) nyelven megfogalmazott kényezetek közötti kapcsolattal foglalkozik. Ez a megfeleltetés képezi a modelltranszformációk futásidejű validálásának alapját, amely magában foglalja mind az egyszerű transzformációs szabályok, mind pedig a teljes transzformációk vizsgálatát. Az értekezés optimalizált algoritmusokat mutat be kényezetek korai kiértékelésére, amelyek hatékonyabb transzformációs folyamatot eredményezik.

A modelltranszformációk validálása a transzformációs szabályokhoz rendelt kényez滕en alapozik. A módszer gyakori következménye, hogy egy-egy kényezt többször is megjelenik egy transzformációban, és a kényeztek átszövik a transzformációs szabályokat. A második eredménycsoport egy aspektus-orientált kényeztkezelési megközelítést tárgyal, amely konzisztens kényeztfeldolgozás mellett a transzformációs szabályokat és magukat a kényezteket is újrafelhasználhatóvá teszi. Emellett az értekezés ismertet egy normalizáló algoritmust, amely navigációs lépések eltávolításával gyorsítja a kényeztek kiértékelését, és ezáltal a teljes transzformációt.

A harmadik eredménycsoport egy vizuális vezérlőnyelvet (Visual Control Flow Language - VCFL) mutat be, amely transzformációs szabályok egymás utáni végrehajtását, kényezshalapú elágazást, hierarchikus szabályokat, párhuzamos szabályfuttatást, iterációt és rekurziót támogat. Ezt követően az értékelés bemutatja a transzformációs szabályok összevonását elősegítő algoritmusokat. Ezek az algoritmusok teszik lehetővé a transzformációk viselkedésének az elemzését. Egy ilyen előre megállapítható tulajdonság például a transzformációk terminálása.

Preface

Dedication

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

Nyalatkozat

Alulirott Lengyel László 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 forrágból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2006 Június

(Lengyel László)

\(^0\)A bírálatok és a védéről készült jegyzőkönyv a későbbiekben a Dékáni Hivatalban elérhetőek.
Acknowledgments

This thesis could not have been created without the support of many people. First of all I am really grateful to my parents and my sister, who have been believing in me and standing by me for all my life. I am very thankful to Dóra because of her support, infinite tolerance and understanding. Furthermore, I would like to thank my friends - the time spent together also helped my work.

I am indebted to Tihamér Levendovszky, who inspired my research, proofread the papers as well as this work. I would like to thank him the long Friday afternoon consultations, his advice and professional support. He has motivated and guided me through this endeavor. This work could not have happened without him.

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, especially his speech in my first year had a significant influence on me.

I am grateful to Gergely Mezei for the common work in the implementation, and I would like to thank him his useful research related questions and remarks. I would like to thank the colleges at the Department of Automation and Applied Informatics (Budapest University of Technology and Economics) their help. I also thank my coauthors for the common work.

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.

Finally, I would like to give thanks to God bringing all these people into my life.
Chapter 1

Introduction

Model-based software development has become one of the most focused software engineering approaches. Model-driven development necessitates the transformation of models between different stages and tools of the design process.

In model-based software development, models and model transformations are elevated to first-class artifacts, there is an increasing need to provide support for techniques and methodologies that are currently missing in the modeling practice. Within a model transformation framework, it is vital to provide foundational support for validation of model transformations.

Model transformations are the heart of model-driven research that assists in the rapid adaptation and evolution of models at various levels of detail [154]. As models represent the system itself, and model transformations make them executable, they need to be analyzed, designed, implemented, checked and maintained as well. For models representing embedded systems that perform critical functions, the importance of correct model transformations is elevated further [66] [190]. An appropriate model transformation approach is needed that can help to assure that the transformation process is reusable and robust [19]. However, in the current model transformation environments, there are few facilities provided for validating transformations in any style. The goal of this thesis is to describe an approach that provides constraint-based, validated model transformation, along with the required constraint management methods, to assist in ensuring the correctness of model transformations.

In general, model transformation techniques can be categorized as either model-to-model transformation or model-to-code translation [36]. Model-to-model transformation translates between source and target models, which can be instances of the same or different metamodels. In a typical model-to-model transformation framework, transformation rules and application strategies are written in a special language that can be either graphical [4] or textual [77]. The source model and the transformation specification are interpreted by the transformation engine to generate the target model. In a model transformation environment - assuming that the model transformation engine works correctly - and the source models are valid inputs, the model transformation specifications are the only artifacts that need to be validated. A transformation specification, like the source code in an implementation, is written by humans and possible sources of errors. Additionally, a transformation specification may be reusable across similar
domains. Therefore, it is essential to ensure the correctness of the transformation. The model transformation approach presented in the current thesis facilitates to specify transformations with high-level constructions and validate them online, during the transformation process. If such a transformation finishes successfully, the result satisfies the conditions required by the high-level constructions.

1.1 Thesis Contributions

This thesis is organized around three concepts that make the metamodel-based visual model transformation flexible, configurable, and efficient.

- Constraint-driven online validated model transformation.
- Strictly controlled visual model transformation.

To illustrate the achieved contributions, the rule editor, control flow and constraint management components of a software framework called Visual Modeling and Transformation System (VMTS) have been developed.

In order to produce the necessary conditions of a framework that facilitates defining and executing validated model transformation, several concepts and related algorithms need to be researched. The relationship between the pre- and postconditions and the OCL constraints enlisted in model transformation rules constitute the principle of the validated transformation. This relation facilitates to specify the properties of the individual transformation rules precisely as well as the whole transformation service. Furthermore, optimization algorithms can provide an efficient constraint evaluation that can be executed during the matching process instead of afterwards. These methods facilitates the early detection of constraint failure, thus they can save significant computational time. In this thesis:

- Propositions are proven to realize the notion of the general validation, preservation, and guarantee for model transformation.
- The constraint-driven, online, validated model transformation is introduced.
- It is shown that in the proposed approach, the successful execution of a transformation means that the generated output artifacts satisfy the required conditions.
- Algorithms are given for the efficient constraint evaluation during the model transformation process.
- It is shown that the computational complexity of the suggested optimizing algorithm reaches the computational complexity of the naive algorithms in the worst case only.
The aspect-oriented constraint management approach solves the problem of crosscutting constraints, and facilitates consistent constraint management in the model transformation rules. In addition, the method makes the constraints as well as the transformation rules reusable. The approach also forms a significant part of the efficient, validated model transformation.

- A method is given to show that using aspect-oriented techniques eliminates the repetitiously appearing constraints from model transformation rules.
- A new type of aspect is proposed: the constraint aspect. A constraint aspect contains not only textually defined conditions but structure, type, weaving, and multiplicity conditions as well.
- It is shown that the OCL constraints can be converted into equivalent constraint aspects.
- Applying the suggested constraint management method not only from individual transformation rules but from whole transformations can be required to validate, preserve or guarantee certain properties.
- The concept of AND/OR clauses is introduced and algorithms are given to normalize constraints as a preprocessor of the efficient constraint management.
- It is proven that the proposed constraint normalization methods do not modify the result of the constraint evaluation. The optimization steps must be executed once for the specified constraints, and their result can be reused any number of times to accelerate the constraint evaluation.
- The proposed normalization method can be applied for arbitrary UML class diagram-based models.
- Weaver algorithms are suggested for aspect-oriented constraint and constraint aspect propagation.
- It is proven that using constraint aspects in model transformation provides the same result as the OCL constraints. Furthermore, it is shown that their structure facilitates a more efficient constraint management and transformation execution.

Visual model transformations developed based on well-defined algorithms often require a strict control over the transformation rules. Therefore, a visual control flow language is provided that facilitates sequencing transformation rules, branching with constraint-driven methods, hierarchical transformation rules, parallel rule execution along with iteration and recursion. Furthermore, the termination properties of the provided transformation language are examined. Thus, in this work:

- A visual control flow language is introduced.
- Algorithms are provided to check the correctness of the transformation models.
Chapter 1. Introduction

- A new method is proposed for metamodel-based model transformation rule composing and self-composing.
- The composing algorithms make the behavior of the transformations predictable.
- Termination conditions are proven for loops, non-exhaustive and exhaustive transformation rules.

As illustrated in this thesis, results related to the enlisted issues constitute an efficient validated model transformation method. The approach supports high-level specification of model-to-model and model-to-code transformations.

1.2 Thesis Structure

The rest of this thesis is organized as follows.

- Chapter 2 is devoted to illustrate the motivations of the results in this thesis. This chapter serves as a general introduction to the topic of metamodeling, constraint management, and model-based development. It also presents a case study to motivate the theoretical results.
- Chapter 3 presents the most well-known graph transformation methods to provide the background and research efforts related to this work.
- In chapters 4, 5 and 6 the contributions of this thesis are presented. Firstly, these chapters introduce the motivations, secondly, describe the related work, and finally, discuss the contributions. The results are divided into three parts: (i) online validated model transformation and optimized constraint evaluation (Chapter 4), (ii) aspect-oriented constraint management (Chapter 5), and (iii) controlled visual model transformation along with its termination properties (Chapter 6).
- In Chapter 7 a synopsis is given on model transformation systems, concentrating on their control flow, attribute transformation, and constraint management support.
- Chapter 8 introduces the rule editor, the control flow and the aspect-oriented constraint management components of the Visual Modeling and Transformation System (VMTS). Furthermore, to illustrate the practical relevance, the application of the results is shown. Out of many possible applications three have been selected: (i) model-based unification of mobile platforms, (ii) model-to-model transformation that generates database representation from class diagrams and emphasizes the importance of aspect-oriented constraints, and (iii) model-to-code transformation that has been chosen to show how software evolution is supported.
- Chapter 9 is devoted to the summary and the outline of future work.
- The Appendix contains (i) some of the detailed proofs (Appendix A), (ii) computational complexity considerations of the constraint evaluation and navigation (Appendix B), (iii) a comparison of model transformation tools in a table format (Appendix C), and (iv) an introduction of verification tools and approaches (Appendix D).
Software system requirements are specified by humans. The specification should be understood by the designer of the model transformation and translated into a given formalism. In model-based software development, such formalism is the OCL constraints propagated to model transformation rules. In general, the offline validation of model transformations is algorithmically hard, therefore, validation is performed online in the proposed approach: during the model transformation process.

To achieve an intuitive and convenient treatment, the expressions of the formalism can be specified with high-level constructs. The designers of the transformation should comprehend and handle both the problem and its specification.

Moreover, the validation introduces a new concern that often crosscuts the transformation rules. To separate these concerns and make them reusable and their management consistent, an aspect-oriented solution is presented for constraint management.

Finally, constraints play an important role in control flows as well. A domain specific visual language is developed to facilitate sequencing the transformation rules, branching between different control flow paths, hierarchical rules, parallel execution of the rules, and iteration along with recursion. The open issues in this case are the validation of the constraints assigned to the transformations, and the termination properties of the control flow. In general, the termination of a graph transformation system is undecidable [166], but in certain cases it is possible to state and prove several termination criteria.

The further part of this chapter is devoted to review the industrial demands and backgrounds which have led to the extensive research of the constraints in validated metamodel-based software model transformation, and the need for efficient constraint management and validated model transformation. An overall view is provided here, as opposed to a narrow concentration on the closely related topics in the following chapters. First, the concepts of metamodeling and the role of constraints in metamodeling are introduced. Then some applications are described where model transformation is a significant and inevitable part of the specific field or technique discussed. Then the open issues are concluded about the necessity of (i) validated model transformation, (ii) efficient and consistent constraint management in model transformation rules and (iii) controlled model transformation. Finally, the chapter summary is given.
2.1 Metamodeling and Constraints

The concept of metamodeling is founded at the beginning of the object-orientation. Metamodeling is an efficient construct for creating a large class of models [12] [13] [14] [189]. For metamodeling tools, it simplifies the creation of generic environments modeling either Unified Modeling Language (UML) [184] or other Domain-Specific Languages (DSLs) [35] that are more specific on a given domain.

Metamodeling is a central technique in the design of visual languages, and it reuses existing domains by extending the metamodel level. Metamodels define the abstract syntax and static semantics of the domain. Metamodels specify the modeling process, how model objects are composed, what attributes they have, what connections can be created between them, what semantics are imposed on them. In the modeling phase, the modeling environment has to apply rules contained by the metamodel.

Once a metamodel is defined, instances of this metamodel, i.e. models which conform to this metamodel can be created. This type-instance relation can be generalized as done in the metamodeling approach where metamodels are instances of meta-metamodels, and so on.

The definition of the metamodel and the metamodeling are as follows.

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

Definition 2.2. The metamodeling is a process describing the creation of the metamodel and the models instantiating the metamodel.

2.1.1 The Unified Modeling Language

Among the modeling languages defined by Object Management Group (OMG) [81], the most known and used is the Unified Modeling Language (UML) [184]. UML is a graphic language to specify, visualize, build, and document the software systems artifacts. Moreover, it provides a standard way to write the models of a system, covering both conceptual elements, like business process and system functions, and the concrete aspects, like classes written in a specific programming language and software components. UML is a standard modeling language standardized by OMG in 1997 as UML 1.1, evolving till the 1.5 version [183], and nowadays there is a UML 2.0 version [184]. While main goal of UML 1.5 is the response to the classic needs of software industry, the UML 2.0 version is a greater evolution in visual modeling, where the new improvements allow to describe many of the new elements found in the software technology of today.

2.1.2 The Object Constraint Language

The Object Constraint Language (OCL) [182] is a formal language for analysis and design of software systems. It is a subset of the industry standard UML that allows software developers to write constraints and queries over object models. OCL is a language that enables to describe expressions and constraints on object-oriented models and other object modeling artifacts. An
expression is an indication or specification of a value. A constraint is a restriction on one or more values of an object-oriented model or system.

Various constraint languages have been used in object-oriented modeling methods (for example Catalysis [27] [50], Syntropy [31] [188]), and programming languages (Eiffel [65] [158]). OCL is set by OMG as a standard for object-oriented analysis and design.

There are four types of constraints: (i) An invariant is a constraint that states a condition that must always be met by all instances of the class, type, or interface. (ii) A precondition to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions. (iii) A postcondition to an operation is a restriction that must be true at the moment that the operation has just ended its execution. (iv) A guard is a constraint that must be true before a state transition fires. Besides these, OCL can be used as a navigation language as well.

As constraints are restrictions on a model or system, they are always coupled to the items used in that model, usually a series of UML diagrams. In fact, the OCL can be used in any model as long as it supports the basic notions of class and instance, attributes, associations and operations.

Software modeling is a synonym for producing diagrams. Most models consist of a number of “nodes and edges”, pictures and some accompanying text. The information conveyed by such a model has a tendency to be incomplete, informal, imprecise, and sometimes even inconsistent. A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to people with a strong mathematical background, but difficult for the average business or system modeler to use. OCL is a formal language that remains easy to read and write.

Many of the flaws in the model are caused by the limitations of the diagrams being used. A diagram simply cannot express the statements that should be part of a thorough specification. OCL expressions are unambiguous and make the model more precise and more detailed. These expressions can be checked by automated tools to ensure that they are correct and consistent with other elements of the model.

The combination of UML and OCL offers the best of both worlds to the software developer. A large number of different diagrams, together with expressions written in OCL, can be used to specify models. To obtain a complete model, both the diagrams and OCL expressions are necessary. Without OCL expressions, the model would be severely underspecified; without the UML diagrams, the OCL expressions would refer to non-existing model elements, as there is no way in OCL to specify classes and associations. Only when we combine the diagrams and the constraints can we completely specify the model.

Similarly to models, model transformation rules also must be specified precisely by constraints. Beyond the topology of the visual model transformation rules, additional constraints must be defined to ensure the correctness of the properties. OCL constraints provide a solution
for the unsolved issues, because these problems can be addressed by constraint validation, but topological transformation methods cannot perform and express these kinds of model properties.

### 2.1.3 Domain-Specific Modeling

Beyond UML there are fields where special languages are needed, since UML would be too general for them. For this reason, Domain Specific Languages (DSLs) [35] continue to play an important role in modeling software and hardware systems.

Domain-Specific Modeling (DSM) languages provide a viable solution for improving development productivity by raising the level of abstraction beyond coding [187]. With DSM, the models are composed of elements representing concepts that are part of the problem domain world, not the code world (unlike for example the core UML concepts). DSM languages follow domain abstractions and semantics, allowing developers to perceive themselves as working directly with domain concepts. In many cases, full final product artifacts can be automatically generated from these high-level specifications with domain-specific code generators. Domain-specific modeling improves and accelerates the software or system development process.

### 2.2 Model-Driven Development

Model transformation has become one of the most focused research field, motivated by for instance OMG’s Model-Driven Architecture (MDA) [181] and Model-Integrated Computing (MIC) [186] [189] [190]. Model-based development (MBD) is an increasingly applied method in producing software artifacts. Model-driven development approaches emphasize the use of models at all stages of system development.

Model-based software development requires the transformation of the models between various stages. These transformations must be precisely specified, which can be accomplished along with constraints enlisted in the transformation rules.

In model-based development, models are used to describe all artifacts of the system, i.e., interfaces, interactions, and properties of all the components that comprise the system. These models can be manipulated in a number of different ways to analyze the system, and in some cases to generate the complete implementation of the system. In order to capture the semantics in an effective manner that is as close as possible to the domain of the developed system, building a domain-specific modeling language is a suitable choice.

In order to synthesize/generate an implementation from the design, one has to bridge the gap between the Domain-Specific Modeling Language (DSML) used in the design process and the semantic domain defined by the underlying software/hardware infrastructure or platform.

### 2.2.1 Model-Driven Architecture

Model-Driven Architecture (MDA) [70] [98] [152] [153] [181] offers a standardized framework to separate the essential, platform-independent information from the platform-dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent
model (PIM), one or more platform-specific models (PSM) including complete implementations, one on each platform that the application developer decides to support. The platform-independent artifacts are mainly UML and other software models containing enough specification to generate the platform-dependent artifacts automatically by model compilers (Fig. 2.1).

MDA development focuses first on the functionality and behavior of a distributed application or system, undistorted by characteristics of the technology platform or platforms on which it will be implemented. In this way, MDA separates implementation details from business functions. Thus, it is not necessary to repeat the process of defining a functionality of the application or a system behavior each time a new technology arises. With MDA, functionality and behavior are modeled only once. Mapping from a PIM through a PSM to the supported MDA platforms is implemented by transformation tools, easing the task of supporting new or different technologies.

2.2.2 Model Integrated Computing

Model Integrated Computing (MIC) is a model-based approach to software development, facilitating the synthesis of application programs from models created using customized, domain-specific program synthesis environments. MIC focuses on models, supports the flexible creation of modeling environments, and helps following the changes of the models. At the same time it facilitates code generation and provides tool support for turning the created models into code artifacts. Metamodeling environments and model interpreters together form the tool support for
MIC. So far MIC is the only methodology, which requires metamodeling environments, model processors and provides a framework for them to cooperate to create computer-based systems in the practice [189].

The MIC development cycle (Fig. 2.2) starts with the formal specification of a new application domain. The specification proceeds by identifying the domain concepts, together with their attributes and inter-relationships using metamodeling. After the domain has been defined, the metamodel of the domain is used to generate a domain-specific design environment (DSDE) through the step called Meta-Level Translation. The DSDE can then be used to create domain-specific designs/models. The next step to synthesize executable code, perform analysis or drive simulators. This is achieved by converting the models into another format such as executable code, input language of analysis tools, or configuration files for simulators. This mapping of the models to another useful form is called model transformation and is performed by model transformers. Model transformers are programs that convert models in a given domain into models of another domain. Note that the result of the transformation can be considered as a model that conforms to a different metamodel: the metamodel of the target.

MIC can be considered as a methodology for domain-specific MDA (DSMDA) where the focus is on developing the MDA process for specific domains. An implementation of DSMDA should consist of a domain-specific modeling environment that allows users to describe systems, using domain concepts. This environment is then used to develop domain-independent models. These models represent the behavior and structure of the system with no implementation details. Such models then need to be converted to domain-specific platform-specific models (DSPSM). DSPSM could either be based on the use of domain-specific libraries and frameworks or they do not have any domain-specific information. It is a term that covers all possible platform-based models.

Transformations appear in many, different situations in a model-based development process. A few representative examples are as follows. (i) Refining the design to implementation; this is
a basic case of PIM/PSM mapping. (ii) Aspect weaving; the integration of aspect models/code into functional artifacts is a transformation on the design [11]. (iii) Analysis and verification; analysis algorithms can be expressed as transformations on the design [10].

One can conclude that transformations in general play an essential role in model-based development, thus, there is a need for highly reusable model transformation tools. These tools must make the model transformation flexible, expressive and validated, furthermore, they should support control flow, constraints, parameter passing between consecutive rules, conditional branching, and should be easy to use.

2.3 Graph Rewriting

Graphs provide an expressive and versatile data representation. Typically, nodes represent objects or concepts, and edges represent relationships among them. Hierarchical relationships can be depicted by node-nesting. Auxiliary information is expressed by adding attributes to nodes or edges. Given the widespread use of graphs as a data representation, it is natural that graph manipulations form the basis of many useful computations. Graph manipulations can be represented implicitly, embedded in a program that, among other things, constructs or modifies a graph. Alternatively, graph manipulations can be represented explicitly, using clearly-delineated graph rewriting rules that modify a host graph. The explicit use of graph-rewriting rules offers several advantages. Graph rewriting provides an abstract and high-level representation of a solution to a computational problem.

Graph rewriting [171] is a powerful tool for graph transformation with a strong mathematical background. A graph rewriting rule is applied to a host graph to replace one subgraph by another. The atoms of the graph transformation are rewriting rules, each rewriting rule consists of a left-hand-side graph (LHS) and a right-hand-side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of LHS in the graph to which the rule is applied (host graph), and replacing this subgraph with RHS. Replacing means removing the elements that are in LHS but not in RHS, and gluing the elements that are in RHS but not in LHS.

Graph transformations have been recognized as a powerful technique for specifying complex transformations that can be used in various situations in a software development process [10] [25] [151]. Many tasks in software development can be formulated using this approach including weaving aspect-oriented programs, application of design patterns [167], and the transformation of platform-independent models into platform-specific models [4].

2.4 A Case Study

To illustrate the motivations on a real world example a case study is provided. The case study is a variation of the “class model to relational database management system (RDBMS) model” transformation (also referred to as object-relational mapping). One should design a database for many of the same reasons that one should design any software application: careful design of
software before implementation improves the quality and reduces the cost. A database design
is often referred to as a data model or schema.

The requirements stated against the transformation that it should guarantee are the following
properties:

- Classes that are marked as non-abstract in the source model should be transformed into
  a single table of the same name in the target model. The resultant table should contain
  one added primary key column, one or more columns for each attribute in the class, and
  one or more columns for associations based on the next rule.

- In general, an association may, or may not, map to a table. It depends on the type and
  multiplicity of the association.
  - Many-to-many (N:N) associations, should be mapped to distinct tables. The primary
    keys for both related classes should become attributes of the association table (foreign
    keys). Foreign keys do not allow NULL values, because a link between two objects
    requires that both of them be known.
  - One-to-many (1:N) associations, using one or more foreign key columns should be
    merged into the table for the class on the “many” side.
  - For one-to-one (1:1) associations, also the foreign key should be buried optionally in
    one of the affected tables.

- Parent class attributes should be mapped into tables created from inherited classes.

- An association class should be transformed based on the multiplicity of the association.
  For N:N association the attributes of the association class become columns of the distinct
  table. For a 1:N or 1:1 the attributes of the association class become columns of the table
  in which the foreign key is buried.

The required rules jointly guarantee that the generated database is in third normal form
[20].

At the implementation level, system validation can be achieved by testing. Various tools
and methodologies have been developed to assist in testing the implementation of a system
(for example, unit testing, mutation testing, and white/black box testing). However, in model
transformation environments, it is not enough that the input model is a valid instance of the
input metamodel and the transformation engine is validated. The transformation specification
should also be validated.

There are only few and not complete facilities provided for testing offline transformation
specifications in an executable style. Also, there are several model transformation environ-
ment provides well-defined interfaces that allow to implement transformations using an optional
object-oriented programming language. Related to the output there is nothing that can be guar-
anteed by these transformations. The transformation should be tested: not only the syntactical
but the semantical correctness is also required. In fact, the testing requires huge efforts, and
even after the testing it is not guaranteed that the transformation produces the semantically
correct output for all valid input. The reason is that there is no real possibility that the testing covers all the possible cases. But, in the case of the case study the following issues should be guaranteed by the transformation:

- Each table has primary key.
- Each class attribute is part of a table.
- Each parent class attribute is part of a table created for its inherited class.
- Each many-to-many association has a distinct table.
- Each one-to-many and one-to-one association has merged into the appropriate tables.
- Foreign keys not allow NULL value.
- Each association class attribute buried into the appropriate table based on the multiplicities of its association.

There is a need for a solution that can validate model transformation specifications: online validated model transformation that guarantees if the transformation finishes successfully, the generated output (database schema) is valid, and it is in accordance with the requirements above.

2.5 Open Issues

Model processing is recognized essential in developing software artifacts, furthermore, model transformation lies at the heart of the model-driven approaches [154] [190]. The main topic of the research is the Validated Model Transformation that can be achieved with constraints propagated to the model transformation rules and applying online methods. Constraint specification requires high-level construction to define pre- and postconditions for individual rules and for whole transformations as well. Applying constraints results that they may crosscut the transformation rules and make them tangled. In order to solve the problem of the crosscutting constraints aspect-oriented methods should be applied that facilitates the separation of concerns and their reuse. Finally, efficient model transformation necessities the existence of controlled model transformation with verifiable termination properties.

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

- **Validated model transformation.** The main goal is to develop the concept of the online validated model transformation. In case of successful execution the resulted artifact must satisfy the conditions required by the transformation.

- **Optimized constraint evaluation.** In order to accelerate the transformation process a method should be established that facilitates efficient constraint evaluation. The method should support the early recognition of constraint failure that can make the whole model processing more efficient.
• **Consistent constraint management.** In a metamodel-based model transformation the core concerns are the transformation rules and the crosscutting concerns are the constraints. It would be beneficial to describe a common constraint in a modular manner, and propagate it automatically to the adequate places.

• **Constraint normalization.** The constraint evaluation consists of two parts. (i) Selecting the object on which the constraint needs to be checked, and (ii) executing the checking. Because of the navigations contained by the constraints, the first part of the evaluation requires more computational time. In order to accelerate the transformation process, the number of the navigations in constraints should be reduced.

• **Controlled model transformation.** Often, transformations must follow well-defined algorithms that require strict control over the whole transformation. There is a need for a high-level visual control flow language that supports rule sequencing, branching, parameter passing, parallel execution, hierarchical rules, and iteration along with recursion.

• **Termination properties.** The termination is an important aspect of the validated model transformations. Termination criteria should be established which work for practical applications.

### 2.6 The Visual Modeling and Transformation System

Visual Modeling and Transformation System (VMTS) [91] [143] is an n-layer metamodeling environment which supports editing models according to their metamodels, and allows specifying OCL constraints. Models and transformation rules are formalized as directed, labeled graphs. VMTS uses a simplified class diagram for its root metamodel ("visual vocabulary").

Also, VMTS is a UML-based [184] model transformation system, which transforms models using graph rewriting techniques. Moreover, the tool facilitates the validation of the constraints specified in the transformation rules during the model transformation process.

The principles of metamodel-based model transformation in VMTS are depicted in Fig. 2.3. The figure describes that the transformation is specified by the VCFL control flow model (Section 6) that defines the exact execution order of the transformation rules. The input model is described by the input metamodel, and the output model by the output metamodel. Both input and output metamodels have an effect on the transformation.

The results discussed in this thesis have been validated in VMTS as a proof-of-concept implementation (Section 8.1).

### 2.7 Chapter Summary

This chapter has reviewed various techniques used in the model-based software development. It has been discussed that the increasing demand for visual languages (VL) in software engineering (such as, Unified Modeling Language - UML; Domain-Specific Languages - DSLs) requires more sophisticated transformation mechanisms for diagrammatic languages.
The need for techniques for model transformations has been recently recognized in the UML community [85] [157] [178]. Graph rewriting-based model transformation is an essential tool for many applications, including transforming abstract design models into concrete implementation models [28], for specification techniques, translation of UML into semantic domains [108], and even for the application of design patterns [94]. The new developments in UML [6] [29] emphasize the use of metamodels and model transformation. Related efforts, such as aspect-oriented programming or intentional programming [97] could also benefit from using transformation techniques based on graph rewriting.

To enhance the development of model transformations that provide dynamic semantics, we need a way to define precisely the operation of these transformations on categories of models, and to generate artifacts that would perform the transformation. However, this task is non-trivial as a model transformation can be required to work with two arbitrarily different domains and perform fairly complex computations. Hence, the specification language needs to be powerful enough to cover diverse needs and yet be simple and usable.

Model transformation can be executed functionally, but there is no mechanism to check it. Dealing with OCL constraints, the validation of the result produced by model transformation can be solved.

By this point, the goals have become clear: we need an online validated model transformation, supported by aspect-oriented constraint management along with a control flow defined visually. Combining metamodel-based model transformation and constraints, the techniques listed above seem to provide a promising direction.

The open issues are addressed by introducing the concepts of general validation, preservation and guarantee to achieve the validated model transformation (Chapter 4), furthermore, the aspect-oriented constraint management (Chapter 5).

Graph grammars and graph transformations have been recognized as a powerful technique for specifying complex transformations. Graph rewriting is a frequently applied technique for model transformation. The next chapter discusses the most well-known graph transformation methods that form formal background of the graph rewriting-based model transformation methods.
Chapter 3

Graph Transformation Approaches

This chapter is intended to present the most well-known graph transformation methods. A software engineering-oriented introduction to graph transformation can be found in [18] and [21]; a deeper analysis of the topic is included in [56] [61] [171]. An annotated bibliography is collected in [64].

A pure imperative technique is to traverse the graph applying the facilities of one or more programming language, and produce the required output. This approach is widely used: it is applied in Intentional Programming (IP) [43] [176] [177] [178] to perform transformations to build the whole program tree from different types of segments. Furthermore, compilers [5] [159] also fall into this category. They traverse the abstract syntax tree in order to optimize or generate the desired output code. Traversing the models is fast. Although the simplicity of the programming is tool-dependent, the idea is straight and widely applied. The code performing the transformation, however, is not so clear and hard to maintain. In case of complex model transformation problems, this approach is really hard to control.

Graph transformations and grammars have been an active topic of research for well over three decades. This research can be classified into two broad categories. The first category, graph grammars, is an extension of textual grammars, and it gave rise to node replacement grammars [160] [171] and hyperedge replacement grammars [82][83]. The second category, researches various mathematical fields such as category theory, set theory and algebra applied to graphs. Originally, it was developed as the natural generalization of Chomsky grammars from strings to graphs, to generate and parse visual languages. Consequently, there are parallel terms between the two fields. Similarly to term rewriting rules, graph transformation rules are referred to as rewriting rules. The prominent works in this area are the double pushout [171], and single pushout [61] along with programmed structure replacement systems [22].

Instead of the graph language approach, we use the mechanism of the individual parsing steps, the rewriting rules, for graph transformation purposes. The graph transformation is defined as a sequence of rewriting rules, where each such rule is a pair of graphs called the left-hand side (LHS) and right-hand side (RHS). The rewriting rule operates as follows: LHS of a rule is matched against the host graph, forming a redex, and the parts of the redex not included in RHS are removed. The resulting graph is called a context graph.
The next step depends on the rewriting approach that is used. There are two fundamentally
different types of adding the elements that occur only in RHS. In case of the gluing approach,
the output graph is formed by gluing RHS along the common vertices. The 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. A typical example
for the connecting approach is the node-label controlled (NLC) \[93\] \[171\] rules. The basic
difference between the two approaches can be captured by dealing with the context elements.
The gluing approach needs the context elements to be specified; the connection approach can
work with unknown context. The connection approach, however, cannot preserve elements in
LHS; thus, all the required elements need to be regenerated. The main problem with the context
element approach is that it cannot specify rules like containing all outgoing edges of a node.

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

The definition of the labeled directed graphs (LDG) based on \[171\] is the following.

**Definition 3.1.** Labeled directed graphs (LDG). Let \(\Omega_V\) and \(\Omega_E\) be two given alphabet for
node and edge labels, respectively. Then the labeled directed graph is a six-tuple: \(G = \langle G_V, G_E, s^G, t^G, lv^G, le^G \rangle\), 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 (\(s^G: G_E \rightarrow G_V\)), which map an edge to its the source
and the target vertex, respectively; and finally, \(lv^G: G_V \rightarrow \Omega_V\) and \(le^G: G_E \rightarrow \Omega_E\), which assign
a label to a vertex and an edge from the appropriate alphabet.

### 3.1 The DPO Approach

According to our experience, the double pushout approach is one of the most user friendly
transformation descriptions for model transformation applications. The main reason for this is
the enforcement of the identification condition, which is found to be according to the intentions
of the transformation modeler. The DPO approach is based on categorical constructs.

The definition of the production rule is given based on \[171\] along with the participants of
a rewriting rule:

**Definition 3.2.** A graph production rule \(p: L \xleftarrow{\text{l}} K \xrightarrow{\text{r}} R\) is composed of a production name \(p\),
and a pair of injective graph morphism: \(l: K \rightarrow L\), and \(r: K \rightarrow R\). The graphs \(L, K\) and \(R\) are
called the left-hand side, the interface graph, and the right-hand side graph of \(p\), respectively.

The DPO approach accomplishes rule firing in two steps: after finding a redex (the part
of the host graph matched by the rewriting rule), the first step removes the elements (vertices
and edges) from the redex which are in the redex, but not in RHS graph. This modified redex
is referred to as interface graph. Then as a second step the elements of RHS graph not in the
interface graph but in RHS graph are glued to the interface graph.
Chapter 3. Graph Transformation Approaches

Fig. 3.1. An illustration of direct derivation: \( L, K, R \) are the left-hand side, the interface and the right-hand side graph; \( G \) and \( H \) are the graphs before and after the rule firing and \( D \) corresponds to \( K \); \( l, r, f, g, m, k, n \) are inclusions.

Fig. 3.2. The direct derivation in the SPO approach

The rewriting rule is characterized by a double pushout. The application of the rules results in a direct derivation of the host graph (Fig. 3.1). The category theory framework provides more flexible and more general background, so the DPO approach can be applied to many graph-like categories. For labeled and directed graphs the existence of the pushout (which is the condition to fire a rule) can be ensured by forcing the gluing condition. The gluing condition consists of two parts. Firstly, the identification condition, which states that different vertices in the rewriting rule cannot match the same vertex in the host graph on deletion. Secondly, the dangling edge condition has to be dealt with as well: if a vertex should be deleted which is connected to an edge that is not inside the redex, the production rule cannot be fired. Unfortunately, this makes impossible to delete a connected vertex without considering its environment. Related to the DPO approach a rather tutorial like description can be found in [32] [51] [52] [60], and a more complete summary in [55] [171].

3.2 The SPO Approach

Another branch of the algebraic graph transformation approaches is the single pushout approach [62] [102] [149] [150] [171]. The single pushout approach uses partial graph homomorphisms to form a single pushout as a rule firing condition in the category of \( \text{Graph}^p \), where labeled directed graphs are the objects, and partial homomorphisms are the arrows. The diagram of the direct derivation according to the SPO approach is depicted in Fig. 3.2.

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

In the DPO and the SPO approaches, the pushout constructions play a basic role. The existence of the pushout is a requirement to fire a rule, and all the proofs are built assuming the
existence of the characteristic pushout construction. Consequently, the results can be applied to any category, which can exhibit the appropriate pushout diagram. These generalizations are called High-Level Replacement Systems (HLR) [58] [59] [61] [63]. The generalization has been applied to the DPO approach especially.
Validated Model Transformation with Efficient Constraint Checking

Model transformation means converting an input model available at the beginning of the transformation process to an output model. Graph transformation is a widely used technique for model transformations. Especially, visual model transformations can be expressed by graph transformations, since graphs are well-suited to describe the underlying structures of models.

To define precisely the transformation rules beyond the topology of the visual models, additional constraints must be specified which ensure the correctness of the attributes, or other properties to be enforced. Dealing with Object Constraint Language (OCL) constraints provides a solution for these unsolved issues, because these problems can be addressed by constraint validation, but topological and attribute transformation methods cannot perform and express these kinds of model properties. OCL as the part of the UML standard has been successfully applied in various problem domains. The use of OCL as a constraint and query language in modeling and model transformation is essential.

This chapter introduces the relation between the pre- and postconditions as well as the OCL constraints assigned to the transformation rules. It discusses the concept of general validation, preservation and guarantee that facilitates the online validation of model transformations. In addition, the chapter provides efficient constraint evaluation algorithms that accelerate the model transformation process. Therefore, the main contribution of the chapter is the constraint-driven efficient validated model transformation.

4.1 Backgrounds and Related Work

An approach has been introduced in [146] - metamodel-based rewriting rules -, where the left-hand side (LHS) and right-hand side (RHS) graphs of the transformation rules are built from metamodel elements. This means that an instantiation of LHS must be found in the host graph instead of the subgraph isomorphic to LHS. This metamodel-based approach facilitates to assign OCL constraints to pattern rule nodes (PRNs) - nodes of the rewriting rules.
Chapter 4. Validated Model Transformation with Efficient Constraint Checking

The graph transformation is defined as an ordered sequence of transformation rules (Chapter 6), in other words we can control the transformation process by sequencing the rewriting rules. In the presented approach the transformation rule is a stereotype of the UML activity state (<< Transformation Rule >>) [146].

4.1.1 Design by Contract

The purpose of contracts [155] is to help us build better software by organizing the communication between software elements through specifying the mutual obligations. Contracts are used to guarantee that these communications occur on the basis of precise specifications of what these services are going to be. For the software to be able to guarantee any kind of correctness and robustness properties, they must know the precise constraints over such communications. In a client/supplier relationship, where the client needs a certain service, and the supplier provides that service, the client can have certain obligations before calling the supplier (for example the passed parameters must have a value from a well-defined range of values). These are preconditions, and they are obligations for the client. In the other direction, we are going to express the conditions that the supplier routine must guarantee to the client on completion of the supplier’s task. That is the postcondition of the contract, specifically, the postcondition of that particular routine. The postcondition is also an obligation for the supplier. Besides the pre- and postcondition the third fundamental element of contracts is the invariant. A class invariant is a condition that applies to an entire class. It describes a consistency property that every instance of the class must satisfy.

4.2 Constraint Validation

Constraints must be defined on the metamodel layer and they are validated on the instance layer. LHS and RHS of the transformation rules are the metamodels of the matched and generated structures [146]. Therefore, metamodels and metamodel-based model transformation rules can contain constraints. The metamodel constraints are validated during metamodel instantiation, while the transformation rule constraints during model transformation.

This section addresses the relationship between the constraints enlisted in metamodel-based transformation rules and the pre- and postconditions. The concepts of general validation, general preservation and general guarantee are introduced. Based on these concepts, if a transformation rule is defined with the help of constraints, and the rule has been executed successfully for the input model, then the generated output model satisfies the constraints required by the transformation rule.

Furthermore, this section summarizes the results related to the constraint validation during the model transformation. It presents the Rule Constraint Validator (RCV) algorithm, the Invariant Analysis (IA) algorithm, the Persistent Analysis (PA) algorithm and the combination of the RCV and PA algorithms that results the Optimized Rule Constraint Validator (ORCV) algorithm.
4.2.1 Validating Metamodel Constraints During Metamodel Instantiation

Metamodel instantiation requires that the instance models fulfill all metamodel rules and constraints. Most of the tools that support constraint validation use constraint interpreters. This means that constraints are interpreted during instantiation just before they are evaluated. In the VMTS approach, constraints are validated by the OCL Module [131] that is not a constraint interpreter, since the validation code is generated based on the metamodel constraints, then a binary is compiled from the generated code [104], and the instance models of the metamodel are checked by compiled binary against the constraints (Fig. 4.1) [122] [123]. With this approach it is not necessary to interpret constraints at each constraint evaluation but only once while generating the validation code.

Multiplicity is also a type of constraint but it is not defined in OCL. The Multiplicity Module checks whether the model contains only those types that are defined in its metamodel, and validates the connections between the model elements.

The Validation Module is realized using the Strategy design pattern [72]. The Strategy pattern helps to define a family of algorithms, it encapsulates each one and makes them interchangeable. Strategy lets the validation algorithm vary independently from the clients that use it. This architecture facilitates that one can attach any number of validation algorithm to the system in the future [131].

4.2.2 Validating Transformation Rule Constraints During Model Transformation

The design by contract principle and model transformation by graph rewriting mechanism have been successfully applied in checking software systems. Thus, revealing the connection between the concepts of pre- and postconditions, the OCL constraints assigned to the transformation rules seem a promising direction.
4.2.2.1 Relation between Pre- and Postconditions and OCL Constraints

For the unified treatment, the basic definitions are given that are mainly based on [155].

Definition 4.1. A Transformation and a Finite sequence of rules consist of $n$ number of transformation rules (where $n > 0$) in an ordered sequence. This sequence defines the execution order of the contained transformation rules. The difference between a transformation and a finite sequence of rules is that a finite sequence of rules it always terminates. A transformation, however, can contain infinite number of steps.

Definition 4.2. A precondition assigned to a transformation rule is a boolean expression that must be true at the moment when the transformation rule is fired. A postcondition assigned to a transformation rule is a boolean expression that must be true after the completion of a transformation rule. If a precondition of a transformation rule is not true, then the transformation rule fails without being fired. If a postcondition of a transformation rule is not true after the execution of the transformation rule, the transformation rule fails.

Definition 4.3. Validation of a property. A transformation rule $S$ validates a property $P$ specified by a boolean expression, when the following condition always holds: if a property $P$ was true before the rule $S$ it remains true after the execution of the rule $S$, and if $P$ is false before the rule $S$, the rule $S$ fails.

Preservation of a property. A transformation rule $S$ preserves a property $P$ specified by a boolean expression, when the following condition always holds: if a property $P$ was false (true) before the rule $S$ it remains false (true) after the execution of the rule $S$.

Guarantee of a property. A transformation rule $S$ guarantees a property $P$ specified by a boolean expression, when the following condition always holds: if a property $P$ was true before the rule $S$ it remains true after the execution of the rule $S$, and if $P$ is false, the rule $S$ changes property $P$ to true.

Definition 4.4. If an OCL constraint is specified in RHS of a transformation rule, $x@pre$ means the value of $x$ immediately before the rule was fired even if the value of $x$ has not changed. Similarly, the $x@preS$ means the value of $x$ immediately before the rule $S$ was fired.

Remark 4.5. The $x@pre$ is used when we refer to the value of the $x$ before the execution of the rule the $x@pre$ is enlisted in, but we use the $x@preS$ if there is a finite sequence of rules and we refer to the value of the $x$ predicting the execution of the rule $S$.

A direct corollary of Def. 4.2 that an OCL expression in LHS is a precondition to the transformation rule, and an OCL expression in RHS is a postcondition to the transformation rule. A transformation rule can be fired if and only if all conditions enlisted in LHS are true. Also, if a transformation rule finished successfully, then all conditions enlisted in RHS must be true [119].

Table 4.1 illustrates the truth table of the validation, preservation and guarantee properties.

Pre- and postconditions defined as OCL constraints and propagated to the transformation rules represents low-level constructions. On the other hand, the validation, preservation and guarantee properties are high-level constructions.

Definition 4.6. A model transformation is validated if satisfies a set of high-level constructions.
Chapter 4. Validated Model Transformation with Efficient Constraint Checking

### Table 4.1. Truth table of the validation, preservation and guarantee properties

<table>
<thead>
<tr>
<th></th>
<th>property P before the rule S</th>
<th>property P after the rule S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>rule S fails</td>
</tr>
<tr>
<td>Preservation</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>Guarantee</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

### Table 4.2. The principles of the constraint pair creation (where $C$ denotes a constraint)

<table>
<thead>
<tr>
<th></th>
<th>LHS side</th>
<th>RHS side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>$C$</td>
<td>$C$</td>
</tr>
<tr>
<td>Preservation</td>
<td>-</td>
<td>(NOT $C$ or $C@pre$) and (C or NOT $C@pre$)</td>
</tr>
<tr>
<td>Guarantee</td>
<td>-</td>
<td>$C$</td>
</tr>
</tbody>
</table>

#### 4.2.2.2 The Constraint Pair Concept

A constraint pair in general means two constraints, one of them is enlisted in LHS and the second is in RHS of a transformation rule. A constraint pair can contain only one constraint, in this case the other one is missing. This means that it is unnecessary to specify constraint in both LHS and RHS side. For example in the case of guarantee property only RHS side constraint has to be specified.

A constraint pair is generated automatically from high-level constructions (an OCL constraint and a required constraint type: validation, preservation or guarantee). Table 4.2 summarizes the fundamental constraint pair creation principles. It presents where and in what form the constraints have to be enlisted to realize the required high-level constructions [128].

#### 4.2.2.3 General Validation

Based on the validation property (Def. 4.3) the concept of general validation can be introduced. The goal of the general validation is the following. If a finite sequence of transformation rules is specified with the help of validation type high-level constructions, and the sequence of transformation rules has been executed successfully for a finite input model, then the generated output model satisfies the required validation type high-level constructions.

**Proposition 4.7.** General validation.

- A rule $S$ validates a property $P$ for an input model $M$ if the property $P$ is enlisted in both pre- and postconditions of the rule $S$, and the rule $S$ has been executed successfully for the model $M$.

**Proof.** ($S^{LHS}$ - is the left-hand side (LHS) of the rule $S$, and $S^{RHS}$- is the right-hand side (RHS) of the rule $S$). Assuming that the property $P$ is enlisted in both $S^{LHS}$ and $S^{RHS}$,
and the rule $S$ has been executed successfully for the model $M$, but the rule $S$ does not validate the property $P$.

This is a contradiction because (i) if the property $P$ is not true, the rule $S$ cannot even be fired. (ii) If the property $P$ is true, the rule $S$ can be fired. The property $P$ is enlisted in $S^{RHS}$, and the rule $S$ has been executed successfully for the model $M$, this means that the property $P$ is true after the execution of the rule $S$, which is equivalent to the definition of the validation (Def. 4.3). Hence, the rule $S$ validates the property $P$. \[\square\]

- If a rule $S$ validates a property $P$ for an input model $M$ without conditions in the rule $S$ for the property $P$, then the property $P$ can be enlisted in $S^{LHS}$ and $S^{RHS}$ without changing the result of the rule $S$ for the model $M$.

Proof. Assuming that the rule $S$ validates a property $P$ for a model $M$ without conditions in the rule $S$ for the property $P$. One enlists the property $P$ in $S^{LHS}$ and $S^{RHS}$, then the result for the rule $S$ is different because of the newly added constraints.

This is a contradiction, because (i) if the property $P$ is false, then the rule $S$ fails in both cases. In the first case if the rule $S$ validates the property $P$ and the property $P$ is false, then the rule $S$ fails (Def. 4.3). In the second case if the property $P$ placed in $S^{LHS}$, and the property $P$ is false, then the rule $S$ fails without being fired (Def. 4.2). (ii) If the property $P$ is true, the rule $S$ can be fired. In the first case the property $P$ remains true after the execution of the rule $S$, because the rule $S$ validates the property $P$. In the second case the property $P$ is enlisted in $S^{LHS}$ and $S^{RHS}$, which is equivalent to the definition of the validation. This means that the result cannot be different. \[\square\]

- A finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ validates a property $P$ for an input model $M$ if the property $P$ is enlisted both in preconditions of the rule $S_0$ and in the postconditions of the rule $S_{n-1}$, and the finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ has been executed successfully for the model $M$.

Proof. Based on the proofs above. \[\square\]

- If a finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ validates a property $P$ for an input model $M$ without conditions in $S_0^{LHS}$ and $S_{n-1}^{RHS}$ for the property $P$, then the property $P$ can be enlisted in $S_0^{LHS}$ and $S_{n-1}^{RHS}$ without changing the result of the finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ for the model $M$.

Proof. Based on the proofs above. \[\square\]

4.2.2.4 General Preservation

Based on the preservation property (Def. 4.3), the concept of general preservation can be developed. If the execution of the finite sequence of transformation rules that is defined with preservation type high-level constructions finishes successfully, then it preserves the property values required by the high-level constructions.
Proposition 4.8. General preservation.

- A rule $S$ preserves a property $P$ for an input model $M$ if the expression $(\text{NOT } P \text{ or } P@\text{pre})$ and $(P \text{ or NOT } P@\text{pre})$ (Def. 4.4) is enlisted as an OCL expression in the postconditions of the rule $S$, and the rule $S$ has been executed successfully for the model $M$.

Proof. Assuming that the expression $(\text{NOT } P \text{ or } P@\text{pre})$ and $(P \text{ or NOT } P@\text{pre})$ is enlisted as an OCL expression in $S^{\text{RHS}}$, and the rule $S$ has been executed successfully for the model $M$, but the rule $S$ does not preserve the property $P$.

This is a contradiction, because the expression $(\text{NOT } P \text{ or } P@\text{pre})$ and $(P \text{ or NOT } P@\text{pre})$ is true if and only if the $P@\text{pre} = P$ holds after the execution of the rule $S$, and this means that the rule $S$ preserves the property $P$.

- If a rule $S$ preserves a property $P$ for an input model $M$ without a postcondition in the rule $S$ for the property $P$, then the expression $(\text{NOT } P \text{ or } P@\text{pre})$ and $(P \text{ or NOT } P@\text{pre})$ as an OCL expression can be enlisted in $S^{\text{RHS}}$, and the rule $S$ also preserves the property $P$ for the model $M$.

Proof. Assuming that the rule $S$ preserves a property $P$ for a model $M$ without postconditions in the rule $S$ for the property $P$. One enlists the expression $(\text{NOT } P \text{ or } P@\text{pre})$ and $(P \text{ or NOT } P@\text{pre})$ as an OCL expression in $S^{\text{RHS}}$, but in this case, because of the newly added constraint, the rule $S$ does not preserve the property $P$ for the model $M$.

This is a contradiction, because in the first case the rule $S$ preserves the property $P$. In the second case the enlisted constraint is equivalent to the definition of the preservation (Def. 4.3), hence, the rule $S$ preserves the property $P$ after the constraint propagation as well.

- A finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ preserves a property $P$ for an input model $M$ if the expression $(\text{NOT } P \text{ or } P@\text{pre}S_0)$ and $(P \text{ or NOT } P@\text{pre}S_0)$ is enlisted as an OCL expression in the postconditions of the rule $S_{n-1}$, and the finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ has been executed successfully for the model $M$.

Proof. Based on the proofs above.

- If a finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ preserves a property $P$ for an input model $M$ without conditions in $S^{\text{RHS}}_{n-1}$ for the property $P$, then the expression $(\text{NOT } P \text{ or } P@\text{pre}S_0)$ and $(P \text{ or NOT } P@\text{pre}S_0)$ as an OCL expression can be enlisted in $S^{\text{RHS}}_{n-1}$, and the finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ also preserves the property $P$ for the model $M$.

Proof. Based on the proofs above.
4.2.2.5 General Guarantee

Similarly to general validation and preservation, the concept of general guarantee can also be introduced with the guarantee property (Def. 4.3).

**Proposition 4.9. General Guarantee.**

- A rule $S$ guarantees a property $P$ for an input model $M$ if the property $P$ is enlisted in the postconditions of the rule $S$, and the rule $S$ has been executed successfully for the model $M$.

  **Proof.** Assume that the property $P$ is enlisted in $S^{RHS}$, and the rule $S$ has been executed successfully for the model $M$, but the rule $S$ does not guarantee the property $P$. The property $P$ is enlisted in $S^{RHS}$, and the rule $S$ has been executed successfully for the model $M$, therefore, the property $P$ is true after the execution of the rule $S$. This is equivalent to the definition of the guarantee (Definition 4.3). This contradicts the assumption, hence the rule $S$ guarantees the property $P$.

- If a rule $S$ guarantees a property $P$ for an input model $M$ without a postcondition in the rule $S$ for the property $P$, then the property $P$ can be enlisted in $S^{RHS}$, and the rule $S$ also guarantees the property $P$ for the model $M$.

  **Proof.** Assume that rule $S$ guarantees a property $P$ for a model $M$ without a postcondition in the rule $S$ for the property $P$. One enlists the property $P$ in $S^{RHS}$, but in this case, because of the newly added constraint, the rule $S$ does not guarantee the property $P$ for the model $M$.

  In the first case the rule $S$ guarantees the property $P$. In the second case the enlisted property $P$ is equivalent to the definition of guarantee (Def. 4.3), hence the rule $S$ also guarantees the property $P$ in the second case.

- A finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ guarantees a property $P$ for an input model $M$ if the property $P$ is enlisted in the preconditions of rule $S_i$ (where $1 \leq i \leq n$), and $S_i, S_{i+1} \ldots S_{n-1}$ preserve the property $P$, or the property $P$ is enlisted in the postconditions of rule $S_i$ (where $1 \leq i \leq n$), and $S_{i+1}, S_{i+2} \ldots S_{n-1}$ preserve the property $P$, moreover the finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ has been executed successfully for the model $M$.

  **Proof.** Based on the proofs above.

- If a finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ guarantees a property $P$ for an input model $M$ without conditions in $S^{RHS}_{n-1}$ for the property $P$, then the property $P$ can be enlisted in $S^{RHS}_{n-1}$, and the finite sequence of rules $S_0, S_1 \ldots S_{n-1}$ also guarantees the property $P$ for the model $M$. 


Proof. Based on the proofs above.

The main advantage of the presented general validation, preservation and guarantee is the validated model transformation.

**Proposition 4.10.** A transformation $T$ contains rules $S_0, S_1 \ldots S_{n-1}$ specified by high-level constructions. If the transformation $T$ has been executed successfully for a finite input model $M$, then the conditions required by the high-level constructions stand for the generated output model $M'$.

*Proof.* The high-level constructions after the propagation become pre- and postconditions (low-level constructions) of the transformation rules. Executing a transformation rule $S$ means the following. First the preconditions of the rule $S$ are evaluated. If the evaluation returns false, the execution of the rule and the whole transformation fails. Otherwise the rule is executed, and finally, the postconditions of the rule $S$ are evaluated on the result of the transformation rule. If all postconditions hold for the generated result, the rule $S$ was successful. Executing the transformation $T$ means executing the rules $S_0, S_1 \ldots S_{n-1} \in T$ based on their defined order. This means that the success of the transformation $T$ automatically means the valid output that satisfies the conditions required by the high-level constructions. Therefore, the modeler’s task is to create proper transformation rules and specify them with high-level constructions. If the execution of the transformation finishes successfully, then it produces a valid result.

### 4.2.2.6 Validation, Preservation and Guarantee Algorithms

After specifying a constraint, it can be applied to an individual rule or to the whole transformation. The pseudo code of the algorithms are the following.

**Algorithm 4.1** Pseudo code of the VALIDATION algorithm

1: **VALIDATION** (VMTSConstraint $C$, VMTSTransformation $T$, VMTSMatch $M$): bool
2: VMTSRule $firstRule = T.getFirstRule()$
3: VMTSRule $lastRule = T.getLastRule()$
4: **AddPreconditionToRule**($firstRule, C$)
5: **AddPostconditionToRule**($lastRule, C$)
6: **return** **ExecuteTransformation**(T, M)

Three parameters are passed to the validation algorithm, the first parameter is a *VMTSConstraint* that contains the constraint one want to validate, the second one is a *VMTSTransformation*, which contains the transformation rules in an ordered sequence, and the third one is a *VMTSMatch* that contains the matched nodes. The validation algorithm selects the first and the last rules from the transformation, and enlists the given constraint as a precondition in the selected first rule and as a postcondition in the selected last rule. This will be proper if the transformation contains only one rule, and also if the transformation contains a finite sequence of rules, because if there is only one rule in the control, the *getFirstRule* and the *getLastRule* queries
Chapter 4. Validated Model Transformation with Efficient Constraint Checking

return the same rule. Finally, the algorithm calls the `ExecuteTransformation` method and retrieves its return value.

The second pseudo code describes the `ExecuteTransformation` method that is used by all three (Validation, Preservation, Guarantee) algorithms.

**Algorithm 4.2** Pseudo code of the `ExecuteTransformation` method

```
1: ExecuteTransformation(VMTSTransformation T, VMTSMatch M): bool
2: for all VMTSRule rule in T.getRulesInOrderedSequence() do
3:   if rule.hasPrecondition then
4:     if not EvaluateConstraints(rule.preconditions, M) return false
5:     rewritingResult = FireRule(rule, M)
6:   if rule.hasPostcondition then
7:     if not EvaluateConstraints(rule.postcondition, rewritingResult) return false
8: return true
```

The `ExecuteTransformation` method applies the transformation rules contained by the given transformation to the passed match. If a rule has a precondition, the method calls the `EvaluateConstraints` method with the preconditions. If the evaluation returns false, then the execution of the rule and the whole finite sequence of rules fails, and the algorithm returns false. Otherwise, the method calls the `FireRule` function, and then if the rule has postconditions, they are validated on the transformation result. If all postconditions hold for the generated result, then the rule was successful.

**Algorithm 4.3** Pseudo code of the `EvaluateConstraints` method

```
1: EvaluateConstraints(VMTSConstraint[] Cs, VMTSMatch M): bool
2: for all VMTSConstraint C in Cs do
3:   destNode = NavigateToDestinationNode(C, M)
4: if not Evaluate(C, destNode) then
5:  return false
6: return true
```

The `EvaluateConstraints` method evaluates the given constraints against the passed match. It iterates over all constraints and navigates to their destination nodes. If any constraint fails the evaluation, the method returns false; otherwise, it returns true.

**Algorithm 4.4** Pseudo code of the Preservation algorithm

```
1: Preservation (VMTSConstraint C, VMTSTransformation T, VMTSMatch M): bool
2: VMTSRule firstRule = T.getFirstRule()
3: VMTSRule lastRule = T.getLastRule()
4: AddPostconditionToRule(lastRule, (NOT C || C@pre firstRule) && (C || NOT C@pre lastRule))
5: return ExecuteTransformation(T, M)
```

The preservation algorithm selects the first and the last rules from the transformation, creates a constraint expression based on the given constraint: `(NOT C or C@pre firstRule) and (C or NOT C@pre lastRule)`, enlists this created expression as an OCL expression in the postconditions of the selected last rule, and finally, it calls the `ExecuteTransformation` method and obtains its return value.
Algorithm 4.5 Pseudo code of the Guarantee algorithm

1: Guarantee (VMTSConstraint C, VMTSTransformation T, VMTSMatch M, int indexOfMarked): bool
2: if (T.Rules.Length == 1 || indexOfMarked < 0) then
3:    AddPostconditionToRule(T.getLastRule(), C)
4: else
5:    AddPostconditionToRule(T.getRuleByIndex(indexOfMarked), C)
6:    for (i = indexOfMarked + 1; i < T.Rules.Length; i++) do
7:        AddPostconditionToRule(T.getRuleByIndex(i), (NOT C || C@pre) && (C || NOT C@pre))
8: return ExecuteTransformation(T, M)

The fourth parameter of the guarantee algorithm - indexOfMarked - contains the index of a marked rule, this rule guarantees the given property, and the rules after this marked rule preserve this property. If the given control has exactly one rule, or if the given indexOfMarked is less than zero (there is no marked rule), the algorithm enlists the given constraint in the postconditions of the last rule (lines 2-3). If the parameter indexOfMarked contains valid index, the algorithm enlists the given constraint in the postconditions of the rule specified by the indexOfMarked (line 5), and enlists the condition (NOT C || C@pre) && (C || NOT C@pre) in postconditions of the rules S_{indexOfMarked+1}, ...S_{n-1} (lines 6-7). Related to the finite sequence of rules, based on the Proposition 4.9 there are two possibilities to require from the rules S_0, S_1 ...S_{n-1} to guarantee a certain property P: (i) enlist the property P in the preconditions of rule S_i (1 ≤ i ≤ n) and require from the rules S_i, S_{i+1} ...S_{n-1} to preserve the property P, or (ii) enlist the property P in the postconditions of rule S_i (1 ≤ i ≤ n) and require from the rules S_{i+1}, S_{i+2} ...S_{n-1} to preserve the property P. Guarantee algorithm uses the latter option (lines 5-7). Finally, the algorithm calls the ExecuteTransformation method and retrieves its return value.

4.2.3 The Rule Constraint Validator Algorithm

Fig. 4.2 presents a block diagram to illustrate how the Rule Constraint Validator (RCV) algorithm validates the constraints contained by the transformation rules during the model transformation (Algorithm 4.7) [129] [131]. Recall that LHS and RHS of a transformation rule are built from the metamodel elements. It is possible that LHS and RHS use different metamodels. The transformation rule contains OCL constraints. The transformation uses matches found by the matching process and the compiled binary [122] [123] to validate the constraints on the matched parts of the input model. If and only if a match satisfies the constraints (preconditions), then the transformation generates the transformation result, and if and only if the transformation result satisfies the postconditions, then the rule was successful.

The complexity to query a VMTSRuleNode or a VMTSRuleEdge from the database is O(4) + O(lg v_r) = O(lg v_r) and O(lg e_r) steps, where v_r and e_r denote the number of the nodes and the edges in the actual transformation. Similarly, obtaining a metamodel (model) node or
edge means $O(lg v_m)$ and $O(lg e_m)$ ($O(lg v)$ and $O(lg e)$) steps, where $v_m$, $e_m$, $v$ and $e$ are the number of the nodes and edges in the actual metamodel and input model [123] [129].

Proposition 4.11. The computational complexity of the RCV algorithm for constraint validating is the following:

- $O(s)$ + $O(lg v_m)$ - once for a transformation rule, and
- $O(lg v)$ - for each rule firing,

where $s$ is the number of the symbols in the OCL constraints, $v_m$ is the number of the metamodel vertices and $v$ is the number of the input model vertices.

The execution of a transformation is presented by the Algorithm 4.7.

4.2.4 The Invariant Analysis Algorithm

It can be assumed that all model conforms to its metamodel, every input model satisfies the constraints defined in the metamodel. This assumption can be made without restricting the generality, because using metamodel-based tools the created models have to conform to their metamodels. Since a transformation rule is built from metamodel elements, the transformation rule can be applied to an input model if and only if the input model and LHS of the transformation rule have the same metamodel, or the metamodel of the input model contains the metamodel of LHS. In other cases it is not possible to find match in the input model because LHS of the transformation rule and the metamodel of the input model contain different types.

Invariant Analysis (IA) means the comparison of the constraints contained by the metamodel and the transformation rule. The goal of the Invariant Analysis is to reveal immediately after the transformation rule specification if a constraint in the transformation rule contradicts a constraint in the metamodel. This is an earlier phase, where it can be explored whether there is any contradiction between the constraints. Invariant Analysis can modify the IsAlreadyChecked property of a constraint in the transformation rule to true if it follows from a constraint contained by the metamodel [129].
Invariant Analysis is an offline algorithm; it is not possible to evaluate pre- and postconditions offline, but those constraints can be checked that use only type and not instance-specific properties.

After the specification of a transformation rule the rule is modified only few times, but it is fired any number of times. Invariant Analysis is independent from the rule firing; it runs only few times for a transformation rule, immediately after the rule creation and modification, therefore the complexity of the Invariant Analysis does not increase the complexity of the transformation process.

Summarizing the Invariant Analysis (Algorithm 4.6), it compares the constraints contained by the metamodel and the transformation rules, and decides which constraints in the transformation rule will be certainly proper and which not.

**Algorithm 4.6 Pseudo code of the InvariantAnalysis algorithm**

1: InvariantAnalysis \((VMTSID \; ruleID, \; out \; VMTSConstraintContradictionList \; contradictionList)\): bool
2: VMTSRuleNode[] ruleNodes = GetRuleNodesByRuleID(ruleID)
3: for all VMTSRuleNode ruleNode in ruleNodes do
4:   VMTSNode typeNode = GetVMTSNodeByID(ruleNode.TypeID)
5:   VMTSConstraintContradictionList currentContradictionList = CompareConstraints(ruleNode.Constraints, typeNode.Constraints)
6:   if currentContradictionList != NULL then
7:     AddToContradictionList(contradictionList, currentContradictionList)
8:   if contradictionList.Count > 0 then
9:     return false
10: else
11: return true

The algorithm InvariantAnalysis queries the PRNs contained by the transformation rule based on the given ruleID. The method affects the PRNs contained by both LHS and RHS. In the for all loop the algorithm queries the metamodel node (typeNode) by the typeID of the actual PRN and calls the CompareConstraints method with the PRN and typeNode constraints. The CompareConstraints method compares the passed constraints, and returns the contradictions if any. If there are returned contradictions between the constraints contained by the metamodel and the transformation rule, the algorithm places them into the contradictionList, which is an output parameter. The return value of the InvariantAnalysis is false if there was at least one contradiction, true otherwise.

The complexity of the GetRuleNodesByRuleID method is \(O(lg v_r)\) and the complexity of the GetVMTSNodeByID method is \(O(lg v_m)\). The for all loop runs for each PRN. The complexity of the method CompareConstraints depends on the number of the passed PRN and the metamodel node constraints. Let \(c_{ri}\) be the number of the constraints contained by the passed PRN \(i\) and \(c_{mi}\) the number of the constraints contained by the metamodel node of the same type. The algorithm needs \(O(c_{ri} * c_{mi})\) time to compare the constraints contained by
the PRN \( i \) with the corresponding constraints in the metamodel node. The complexity of the complete for all loop is \( O(\sum_{i=1}^{v_r} (\lg v_m + (c_{ri} * c_{mi}))) \).

If the Invariant Analysis is used during the transformation rule creation, and the algorithm detects that at least one of the constraints in the transformation rule contradicts a constraint in the metamodel, then the algorithm marks the contradicting constraint. Since the rule would be unsuccessful, and therefore the saved computational time is \( SC_{IA} \geq n^k - C_{IA} \). \( SC_{IA} \) denotes the saved computational time using Invariant Analysis, \( n^k \) is the complexity of the matching process (where \( n \) is the number of the input model nodes, and \( k \) is the number of nodes in the matched submodel [144] [171]) and \( C_{IA} \) is the computational complexity of the Invariant Analysis.

Proposition 4.12. The computational complexity of the Invariant Analysis is \( O(\lg v_r) + O(\sum_{i=1}^{v_r} (\lg v_m + (c_{ri} * c_{mi}))) \). The computational time we can save using the Invariant Analysis is \( SC_{IA} \geq n^k - C_{IA} \).

The saved computational time presented here is valid only for the first rule firing, because we can assume that after the warning the designer of the transformation will correct the wrong constraint. If the constraint is not corrected, then the execution of the rule is interrupted and the saved computational time is \( n^k \) at each attempt to fire the rule.

4.2.5 The Persistent Analysis Algorithm

The Persistent Analysis (PA) algorithm evaluates the constraints continuously during the matching process. Every constraint propagated to a transformation rule has a property \( \text{IsAlreadyChecked} \), this property is false at the beginning of the matching process (except if the Invariant Analysis algorithm changed it to true). If the Persistent Analysis algorithm has all information to evaluate a constraint (precondition) then checks it, and if the constraint holds then sets its \( \text{IsAlreadyChecked} \) property to true, otherwise if the constraint does not hold, the rule fails. The transformation starts after the matching process. The first step of the transformation is the evaluation of all the preconditions in the match, but in fact the algorithm has to check only the constraints with false \( \text{IsAlreadyChecked} \) value. Persistent Analysis accelerates the whole algorithm (matching and transformation), because if it is revealed in the matching phase that one of the preconditions does not hold, then we can leave the current branch, backtrack, and continue the matching from another position. This saves the computational time of finishing matching at the given position and the time having elapsed between the start of the transformation and finding the contradiction.

The computational complexity of the constraint evaluation during the matching process is at most \( O(\sum_{i=1}^{k} (c_{ri} + c_i)) \), where \( k \) is the number of nodes in the matched submodel, \( c_{ri} \) is the number of the constraints contained by the PRN that is matched to the input model node \( i \), and \( c_i \) is the number of constraints contained by the type node of the matched input model node \( i \). The expression at most in the previous sentence refers to the worst case: the algorithm does not find a contradiction, and has to evaluate all constraints. Otherwise if it finds an unsatisfied
constraint then it does not have to continue the constraint evaluation, and the computational complexity is less. In this algorithm, we can also assume the case that the host graph instantiates its metamodel, and in this case the computational complexity is $O(\sum_{i=1}^{k} c_{ri})$. While each PRN contains a constant number of constraints, the computational complexity in fact is $O(k)$. Some measurement results are depicted in Table 4.3. The saved computational time depends on the time, when the algorithm finds the first not satisfied constraint. If all preconditions are satisfied, the algorithm does not find a contradiction, and the saved computational time comes only from the fact that the constraint evaluation is faster during the matching because the host nodes are directly available. If there is at least one unsatisfied constraint then the saved computational complexity is the complexity of the unexecuted part of the matching algorithm. If we find a contradiction at the beginning, the saved computational complexity is near $n^k$, where $n$ is the number of the input model nodes, and $k$ is the number of nodes in the matched submodel.

**Proposition 4.13.** The computational complexity of the constraint evaluation during the matching process (Persistent Analysis algorithm) is at most $O(k)$, where $k$ is the number of nodes in the matched submodel. In addition, the Persistent Analysis algorithm does not increase the complexity of firing a rule. Using the Persistent Analysis algorithm, the saved computational time is $n^k - r$ if the algorithm finds an unsatisfied constraint, while it evaluates the constraints on the matched input model node $r$.

### 4.2.6 The Optimized RCV Algorithm

The RCV algorithm evaluates constraints after the matches are found, the optimization takes advantage of the fact that it is possible to evaluate certain constraints during the matching process. The Optimized RCV (ORCV) algorithm completes the RCV algorithm with the Persistent Analysis algorithm that results a more powerful constraint validation algorithm.

The next proposition states that the computational complexity of the Optimized RCV algorithm reaches the computational complexity of the RCV algorithm in the worst case only.

**Proposition 4.14.** The computational complexity of the Optimized RCV algorithm never exceeds the computational complexity of the RCV algorithm.

- $C_{ORCV} = C_{RCV}$ - The computational complexity of the ORCV algorithm equals the computational complexity of the RCV algorithm -, if the Persistent Analysis algorithm cannot evaluate any constraint during the matching process.
Chapter 4. Validated Model Transformation with Efficient Constraint Checking

Algorithm 4.7 Pseudo code of the OptimizedRCV algorithm

1: OptimizedRCV(VMTSTransformation $T$, VMTSModel $inputModel$): bool
2: for all VMTSRule $rule$ in $T.getRulesInOrderedSequence()$ do
3: VMTSMatch $M = \text{NewMatch}()$
4: while $M$ is not complete or all branch is tried do
5: VMTSNode $matchNodeCandidate = \text{GetNextMatchNodeCandidate}(M, rule, inputModel)$
6: rule.hasPrecondition and not $\text{EvaluateConstraints}(rule.preconditions, matchNodeCandidate)$ then continue
7: $\text{AddNodeToMatch}(M, matchNodeCandidate)$
8: if $M$ is not complete then return false
9: rewritingResult = $\text{FireRule}(rule, M)$
10: if rule.hasPostcondition then
11: if not ($\text{EvaluateConstraints}(rule.postcondition, rewritingResult)$) then
12: return false
13: return true

1. The computational complexity of the ORCV algorithm is at least $p \times \log v$ times less than the computational complexity of the RCV algorithm if the Persistent Analysis algorithm can evaluate $p$ constraints during the matching process, and all of the evaluated constraints are satisfied. In this case the saved computational time is $SC_{ORCV} \geq p \times \log v$. The saved computational time comes from the fact that the nodes on which the algorithm has to evaluate the constraints are available during the matching process.

2. The computational complexity of the ORCV algorithm is at least $n^{k-r}$ times less than the computational complexity of the RCV algorithm if the Persistent Analysis algorithm finds an unsatisfied constraint, while evaluates the constraints on the matched input model node $r$.

Remark 4.15. The formula contains the $\geq$ operator, because if the constraints contain navigation steps and the input model nodes, which are part of the navigation path, are already matched, then they are directly available.

4.3 Example for Validated Model Transformation

The practical relevance of the results is presented on the transformation Class2RDBMS (Section 2.4) that is a variation of the object-relational mapping.

The whole control flow model of the transformation is presented in Fig. 6.1, and its first transformation rule is depicted in Fig. 4.3. The rule CreateTable processes non-abstract classes and creates tables into the output model. A resultant table has the same name as the class in the input model. The table contains one added primary key column, and one column for each attribute in the processed class. Furthermore, this rule links each non-abstract class to their created table. Using this link, the following rules can reach the appropriate table from the class.
to add new columns and foreign keys to it, based on the adjacent non-abstract and abstract classes.

To require certain properties from the transformation rule the following constraints are applied:

\begin{verbatim}
context Class inv NonAbstract:
  not self.abstract
\end{verbatim}

The constraint \textit{NonAbstract} is assigned to the PRN \texttt{Class} in LHS of the rule \textit{CreateTable}. This link forms a precondition, it requires the rule to process only non-abstract classes. The constraint also propagated to PRN \texttt{Class} in RHS as a postcondition of the rule. The \textit{validate} type constraint means that the value of the attribute \texttt{abstract} must be true both before and after the rule execution.

\begin{verbatim}
context Table inv PrimaryKey:
  self.columns->exists(c | c.datatype = 'int'
  and c.is_primary_key)
\end{verbatim}

The constraint \textit{PrimaryKey} is a postcondition of the rule \textit{CreateTable}, it is assigned to the PRN \texttt{Table}. This \textit{guarantee} type constraint requires the rule that all created table has a primary key of \texttt{int} type.

\begin{verbatim}
context Table inv PrimaryAndForeignKey:
  not self.columns->exists(c | (c.is_primary_key or
  c.is_foreign_key) and c.allows_null)
\end{verbatim}

The constraint \textit{PrimaryAndForeignKey} of \textit{guarantee} type is also a postcondition that necessitates the primary and foreign key columns do not allow NULL values.

\begin{verbatim}
context Atom inv ClassAttrsAndTableCols:
  self.class.attribute->forAll(self.table.column->
  exists(c | (c.columnName = class.attribute.name))
\end{verbatim}
The guarantee type constraint `ClassAttrsAndTableCols` is linked to the PRN `TableHelperNode`, it requires the transformation rule that each class attribute has a created column with the same name in the resultant table.

The transformation rule can be fired if all the constraints enlisted in LHS of the rules hold for the matched part of the input model. In the case of the rule `CreateTable`, the matched class should be non-abstract. The rule is successful if all the postconditions hold for the result of the rule. Similarly, constraints are propagated to the transformation rule `ProcessAssociation` that enforce the correctness of the association processing. The constraints assigned to the transformation rules guarantee the followings from the requirements enlisted in Section 2.4:

- Each resultant table contains a primary key of `int` type.
- Each class attribute has generated table column.
- Primary and foreign keys do not allow NULL values.
- Each many-to-many association has a distinct table.
- Each one-to-many and one-to-one association is merged into the appropriate tables.
- Each association class attribute buried into the appropriate table based on the multiplicities of its association.

As it is presented, after a successful rule execution the conditions hold and the output is valid that cannot be achieved without constraints.

A more efficient solution would be if the constraints that require the presented properties are propagated automatically to the transformation rules creating or modifying tables. The next chapter provides a solution for it.

### 4.4 Chapter Summary

The metamodel-based specification of the transformation rules allows assigning OCL constraints to the rules, and they are able to express local constraints. However, this does not mean that evaluating them does not involve checking other model elements in the input model. OCL constraints enlisted in the rules have effect on the instances of these rules, on the matched and the replaced submodels.

The relationship between the pre- and postconditions and OCL constraints have been shown, and how the OCL constraints enlisted in the transformation rules can be used to check validation, preservation and guarantee properties, or simply how to validate models with the help of metamodel-based model transformation.

In this chapter, the concepts of general validation, general preservation, and general guarantee have been discussed. It is presented that if a transformation rule defined with the help of high-level constructions, and the rule has been executed successfully for the input model, then the generated output model satisfies the conditions required by the high-level constructions.
Chapter 4. Validated Model Transformation with Efficient Constraint Checking

This means that the modeler’s task is to create rules and specify them with high-level constructions, and if the execution of the transformation rule finishes successfully, it automatically means that the result is valid.

The main limitation of the presented method is the local-nature of the rules. If one wants to specify a constraint for an element, it must be included in a transformation rule, or it must be referenced by the OCL traversal expressions assigned to the rule elements. Consequently, this method does not provide an easy way to check global constraints such as deadlock examination or database is in third normal form. Therefore, those parts of the models that are not affected by a transformation rule constraints cannot be specified with this method. But there are numerous cases, for example, source code generation from statechart model [118] [121] [123], or user interface generation from resource model [68] [69] [141] [142], where the whole right side is generated, thus all the output model elements can be validated.

An important part of the constraint validation method is that the approach does not interpret the constraints. It generates source code and compiles it to a binary that validates the metamodel and transformation rule constraints. This method facilitates that the complexity of the presented constraint validation method can be exactly determined.

Finally, in this chapter, algorithms are provided to validate the graph rewriting-based model transformation with the help of the constraints contained by the transformation rules. The Invariant Analysis algorithm is applicable for both LHS and RHS constraints, while the Persistent Analysis algorithm affects only the constraints in LHS (preconditions of the transformation rule). We have also been discussed the computational complexities of the presented algorithms and the computational time that we can save during the model transformation using these algorithms [129].
Chapter 5

Aspect-Oriented Constraint Management

The previous chapter has introduced the relation between the pre- and postconditions and the constraints propagated to transformation rules. These constraints ensure the correctness of the attributes, and enforce other properties of the transformation rules.

In many cases not only a transformation rule but a whole transformation is required to validate, preserve or guarantee a certain property. To meet this expectation, all the transformation rules need to be taken into consideration. If one defines a constraint written in OCL for several transformation rules or for a whole transformation, then the same constraint appears numerous times in the transformation and crosscuts it. The modification and deletion of a crosscutting constraint is not consistent, because such an operation must be performed on all of its occurrences. Moreover, it is often difficult to reason about the effects of a complex constraint when it is scattered across the numerous nodes in transformation rules [109] [120] [128] [130] [141].

In metamodel-based model transformation the core concerns are the transformation rules and the crosscutting concerns are the constraints. It would be beneficial to describe a common constraint in a modular manner, and propagate it automatically to the adequate places.

Aspect-Oriented Software Development (AOSD) [67] [87] provides a technique to address separation of concerns (SoC). The methods of AOSD facilitate the modularization of crosscutting concerns within a system. AOSD attempts to decompose a system further into core application concerns. A crosscutting concern cannot easily be modularized, because the concern tends to be scattered across a system. AOSD allows crosscutting concerns to be separated from the core concerns, thus one can reason about them in isolation, and weave them back later to form a fully functioning system.

This chapter discusses an aspect-oriented approach that can be used to solve the problem of crosscutting constraints in metamodel-based transformation rules. Firstly, the chapter gives a general overview about the aspect-oriented techniques and their purpose. Secondly, the constraint management problem and a method to define constraints as aspects separately from transformation rules are introduced. A new type of aspect referred to as constraint aspect is proposed. Besides the textual conditions, a constraint aspect also contains a structure that
specifies the weaving process. An algorithm is given that facilitates to create constraint aspects from OCL constraints. Furthermore, the weaving aspect that supports the constraint propagation process is introduced. It facilitates the type- and property-based weaving. An optimization algorithm is given to normalize constraints propagated to UML class diagram-based models. In addition, weaver algorithms are provided to propagate aspect-oriented constraints and constraint aspects to model transformations.

The approach presented here makes it possible to define constraints separately from the transformation rules and facilitates specifying their propagation to the model transformation rules. This method gives an efficient constraint management, solves the problem of crosscutting constraints, makes the constraints and the transformation rules reusable, in addition, it supports the validated model transformation. Furthermore, the approach also facilitates to validate not only individual rules but also whole transformations.

5.1 Backgrounds and Related Work

5.1.1 Aspect-Oriented Software Development

Aspect-oriented (AO) techniques are popular today for addressing crosscutting concerns in software development. Aspect-oriented software development (AOSD) [67] [87] methods enable the modularization of crosscutting concerns within software systems. AOSD techniques and tools, applied at all stages of the software lifecycle, are changing the way in which software is developed in various application domains, ranging from enterprise to embedded systems.

A growing area of research in the field of software development is focused on bringing aspect-oriented techniques into the scope of analysis and design [67] [163] [164]. The motivation of these efforts is the systematic identification, modularization, representation, and composition of crosscutting concerns such as security, mobility, distribution, and resource management. Its most important potential benefits include improved ability to reason about the problem domain and the corresponding solution, reduction in the size of software model and application code, development costs and maintenance time, improved code reuse, architectural and design level reuse as well as better modeling methods across the lifecycle.

AOSD is a technology that extends the separation of concerns (SoC) in software development. Aspects may appear in any stage of the software development lifecycle (e.g. requirements, specification, design, and implementation). Crosscutting concerns can range from high-level notions of security to low-level notions such as caching, and from functional requirements such as business rules to non-functional requirements like transactions. Research in AOSD is driven by the fundamental goal of better separation of concerns.

AOSD is an emerging paradigm that provides explicit abstractions for concerns that tend to crosscut over multiple system components and result in tangling in individual components. By representing crosscutting concerns or aspects as first-class abstractions, and by providing new composition techniques for combining aspects and components, the modularity of the system can be improved, leading to a reduced complexity of the system and easier maintainability.

The aim of this section is to provide a conceptual discussion on the problems that are tackled by AOSD. The discussed concepts and problems appear to be general for the complete
software development life cycle. The following important issues are introduced: the concerns and their separation, the problem of the crosscutting and tangling concerns, the aspect-oriented decomposition, and aspect weaving. Furthermore, aspect-oriented programming and aspect-oriented modeling is presented in more detail.

5.1.1.1 General AOSD Concepts

Separation of concerns. To understand the ideas in AOSD, first we have to look at the separation of concerns principle, which can be actually considered one of the key principles in software engineering. This principle states that a given problem involves different kinds of concerns that should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability, maintainability, and reusability. The separation of concerns principle is a ubiquitous software engineering principle, which can be applied in various ways [44].

Concerns. Despite a common agreement on the necessity of the application of the separation of concerns principle, there is not a well-established understanding of the notion of concern. For example, in object-oriented methods the separated concerns are modeled as objects and classes, which are generally derived from the entities in the requirement specification and use cases. In structural methods, concerns are represented as procedures. In aspect-oriented programming, the term concern is extended with the crosscutting properties such as synchronization, memory management and persistency. In a sense one can consider this as a generalization of the notion of concern in the context of programming languages. Although this issue is considered a natural development, it increases the necessity of renewed understanding of what concerns are, because concerns are not anymore restricted to objects and functions. Moreover, the task of separating the right concerns is complicated, because now one has to deal with a larger set and variety of concerns.

Any engineering process has many issues to deal with [67]. These range from high-level requirements ("The system shall be stable") to low-level implementation issues ("Remote values shall be cached"). Some concerns are localized to a particular place in the emerging system ("When the Ctrl+A hot key is pressed, a defined window shall pop up"), some involve systematic behavior ("All exception handling shall be traced"). Generically, all these are called concerns, though AOSD technology is particularly directed at the last, systematic class.

Problem Statement

Modular Decomposition. The separation of concerns principle states that each concern of a given software design problem should be mapped to one module in the system. Otherwise, the problem should be decomposed into modules such that each module has one concern. The advantage of this is that concerns are localized and as such can be easier understood, extended, reused, and adapted. This decomposition process is illustrated in Fig. 5.1a. The design problem is decomposed into concerns \(C_1, C_2 \ldots C_n\) and each of these concerns is mapped to a separate module \((M_1, M_2 \ldots M_n)\). A module is an abstraction of a modular unit in a given design language (e.g. class or function) [7].
Crosscutting Concerns. Many concerns can indeed be mapped to single modules. Some concerns, however, cannot be localized and separated easily, and given the design language we are forced to map such concerns over many modules. This is called crosscutting. In Fig. 5.1b, for example, concern $C_2$ is mapped to the modules $M_1$, $M_2$ and $M_{n-1}$. We say that $C_2$ is a crosscutting concern or an aspect. Examples of aspects are e.g. tracing, synchronization, and load balancing. Aspects are not the result of a bad design but have more inherent reasons. A bad design including mixed concerns over the modules could be refactored to a neat design in which each module only addresses a single concern. However, if we deal with these crosscutting concerns, this is not possible in principle that is, each refactoring attempt will fail and the crosscutting will remain. A crosscutting concern is a serious problem, since it is harder to understand, reuse, extend, adapt and maintain the concern because it is spread over many places. Finding the places where the crosscutting occurs is the first problem, adapting the concern appropriately is another problem.

Things may even worsen if there are multiple crosscutting concerns. For example in Fig. 5.2a there are two crosscutting concerns $C_2$ and $C_3$.

Tangled Concerns. Since crosscutting concerns cannot be easily localized and separated several modules will include more than one concern. Concerns are tangled in the corresponding module. For example in Fig. 5.2a, the concerns $C_2$, $C_3$ and $C_{n-1}$ are tangled in the module $M_{n-1}$. Note that concern $C_{n-1}$ is not crosscutting.

Join points. In Fig. 5.2b the same information is depicted as in Fig. 5.3. The modules are aligned vertically whereas the concerns horizontally. The circles represent the places where the concerns crosscut a module. These are called join points. Join points can be at the level of a module (class) or be more refined and deal with subparts of the module (e.g. attribute or operation). Crosscutting can be easily identified if a concern is followed in a vertical direction (multiple join points). Tangling can be detected if each module is followed in the horizontal direction.
From this point, the AOSD concepts are introduced based on the AspectJ [9] terminology.

**Aspect.** Obviously, a given design problem can have crosscutting concerns and conventional abstraction mechanisms that cannot be handled appropriately with these concerns. AOSD provides explicit abstractions for representing crosscutting concerns referred to as aspects. As such, a given design problem is decomposed into concerns that can be localized into separate modules and concerns that tend to crosscut over a set of modules.

**Pointcut specification.** To specify the points that the aspect crosscuts, a pointcut specification is used. A pointcut specification is, essentially, a predicate over the complete set of

---

**Fig. 5.2.** (a) Crosscutting concerns ($C_2$ and $C_3$), (b) Join points, Tangling and Crosscutting concerns

**Fig. 5.3.** Aspect separation
join points that the aspect can crosscut. A pointcut specification can enumerate the join points or provide a more abstract specification. In an abstract sense, aspects can thus be specified as follows.

\[
\begin{array}{ll}
\text{Aspect name} & \\
\text{Pointcut specification} & \\
\end{array}
\]

**Advice.** The crosscutting is actually localized in the pointcut specification. The pointcut specification indicates which points the aspect crosscuts but it does not specify what kind of behavior is needed. For this the concept of advice has been introduced. An advice is a behavior that can be attached before, after, instead of or around a join point in the pointcut specification.

\[
\begin{array}{ll}
\text{Aspect name} & \\
\text{Pointcut specification} & \\
\text{Advice} & \\
\end{array}
\]

**Weaving.** Having separated the aspects, their management (e.g. maintenance or reuse) become easier and consistent. In order to obtain a complete system from the separated artifacts, AO provides the weaving mechanism. Weaving is the process of composing the core functionality modules with aspects, thereby yielding a working system. The various AO approaches have defined several different mechanisms for weaving.

There are several aspect-oriented approaches and languages. Although they differ in the way of specifying aspects, pointcuts, advices, and weaving basically adopt the concepts presented above.

In summary, AOSD emphasizes the separation of concerns and is designed to handle complex structures. Both aspect-oriented programming (AOP) and aspect-oriented modeling (AOM) are part of the AOSD paradigm.

### 5.1.1.2 Aspect-Oriented Programming

The history of programming has been a slow and steady climb from the depths of direct manipulation of the underlying machines to linguistic structures for expressing higher-level abstractions. The progress in programming languages and design methods has always been driven by the invention of structures that provide additional modularity. Subroutines assembled the behavior of unstructured machine instructions, structured programming argued for semantic meaning for these subroutines, abstract data types recognized the unity of data and behavior, and object-orientation (OO) generalized this to multiplicity of related data and behaviors.

The current state-of-the-art paradigm in programming is the object-oriented technology. With objects, the programmer is supposed to think of the universe as a set of instances of particular classes that provide methods, expressed as imperative programs, to describe the behavior of all the objects of a class.

Object-orientation has many advantages, particularly in comparison with its predecessors. Objects facilitate modularization. The notion of sending messages to objects helps concentrate on the programmer’s thinking and aids understanding code. Inheritance mechanisms in object
systems provide a way both to assign the related behaviors to multiple classes and to make exceptions to that regulation.

Objects are not the last improvement in programming paradigms. AOSD techniques are the next step in this progression. Aspects introduce new linguistic mechanisms to modularize the implementation of concerns. Each of the earlier steps (with the minor exception of multiple inheritance in OO systems) focused on centralizing on a primary concern. AO, like its predecessors, is about recognizing that software systems are built with respect to many concerns and those programming languages, environments, and methodologies must support modularization mechanisms that honor these concurrent concerns. AO is technology for extending the kinds of concerns that can be modularized separately and efficiently [67].

AOSD has started on programming level. During the last decade several aspect-oriented programming languages were introduced. Some of the prominent aspect languages are AspectJ [9], AspectC++ [8], HyperJ [92], ComposeJ [30], and DemeterJ [40]. AspectJ is a general purpose extension to Java that introduces new language constructs to represent and compose aspects. AspectC++ extends the AspectJ approach to C/C++. It is a set of C++ language extensions to facilitate aspect-oriented programming with C/C++. HyperJ supports multi-dimensional separation of concerns for Java and operates on standard Java class files and produces new class files to be used for execution. In ComposeJ, aspects are represented through composition filters that are declaratively specified and integrated in the corresponding programming language. DemeterJ is an aspect-language that supports the encapsulation of traversal-related behavioral aspects in Java.

Aspect-oriented programming is a technology for separation of crosscutting concerns on the programming language level into single units referred to as aspects. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviors that affect multiple classes into reusable modules. Aspectual requirements are concerns that introduce crosscutting in the implementation. Typical aspects are synchronization, error handling or logging. With AOP, each aspect can be expressed in a separate and natural form, and can be automatically combined together into a final executable form by an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of procedures, modules or objects, increasing the reusability of the code. The differences between AOP and traditional programming are shown in Fig. 5.4. Compared to traditional approaches AOP allows the separation of crosscutting concerns at source code level. The aspect code and the other part of the program can be woven together by an aspect weaver before the program is compiled into an executable.

5.1.1.3 Aspect-Oriented Modeling

Modeling is a key tool in software engineering, allowing the software production process to be represented at a variety of stages and levels of detail. Aspect-oriented modeling techniques allow system developers to address crosscutting and quality objectives, such as security, separately from core functional requirements during system design [74]. An aspect is a pattern of structure and behavior such that it is a crosscutting realization of common structural and behavior characteristics [184].
An aspect-oriented design model consists of a set of aspects and primary models. An aspect model describes how a single objective is addressed in the design, while the primary model addresses the core functionality of the system as given by the functional requirements.

In AOM, weaving rules are used to weave aspect models with the primary model. These rules are stored separately from the aspect and the primary model, which makes both the aspect models and the rules reusable. The aspects and the primary model are composed before implementation or code generation. The composed model supports design analysis. Composition is most often performed manually, but there are tools that automate a part of the composition. Fig. 5.5 gives an overview of AOM.

An aspect-oriented approach is introduced in [78] for software models containing constraints, where the dominant decomposition is based upon the functional hierarchy of a physical system. This approach provides a separate module for specifying constraints and their propagation. A new type of aspect is used to provide the weaver with the necessary information to perform the
propagation: the strategy aspect. A strategy aspect provides a hook that the weaver may call in order to process the node-specific constraint propagations.

Constraint-Specification Aspect Weaver (C-SAW) [15] [75] [79] [80] [200] is an aspect-oriented approach to modeling. The C-SAW weaver framework serves as a generalized transformation engine for manipulating models. C-SAW is a plug-in for the Generic Modeling Environment (GME) [106] [107]. The result of model weaving is a new model that contains adaptations that are spread across the model hierarchy. These adaptations can be undone, and new concerns can be woven. The weaver can be used to distribute any system property endemic to a specific domain across the hierarchy of a model. A weaver can also be used to instrument structural changes within the model according to the dictates of some higher-level requirement that represents a crosscutting concern.

5.2 The Constraint Management Problem

In model transformation, the dominant decomposition is the functional behavior of the transformation rules. The constraints ensure the correctness of the transformation only if they are well-defined by the designer. Although they are responsible for the correctness, the constraints are usually specified after the first draft of the transformation, and treated with secondary importance. They crosscut the transformation, and it is almost impossible for the designer to perform the intuitive rules of verifying the transformation.

Often the whole transformation is required to validate certain model properties. In that case all the transformation rules need to be taken into account [109] [120] [128] [130] [141]. For instance, assume a transformation that modifies the properties of the Person type objects and we would like the transformation to validate that the age of a Person is always under 200 ($\text{Person.age} < 200$). It is certain that the transformation preserves this property if the constraint is defined for all PRN of type Person. This means that the constraint can appear in each transformation rule several times. Therefore, the constraint crosscuts the whole transformation, and a concern that $\text{Person.age} < 200$ crosscuts the functional concerns, for example that the Employee changes its job.

Fig. 5.6 introduces a metamodel and a transformation with two transformation rules and crosscutting constraints. The first transformation rule changes the job of the person Employee. The second rule employs the person ToEmploy at the company NewCompany, furthermore, marries the persons Employee and ToEmploy. The const_age constraint appears at several places in this transformation. Fig. 5.6 shows a concrete case where a constraint is present in many places of the two transformation rules. Another example constraint could be that we require the transformation to preserve that the number of employees of a Company is always between 50 and 300 (50 $\leq$ Company.NumbeOfEmployees $\leq$ 300). These problems also appear in technical applications, such as generating user interface from resource model and user interface handler source code from statechart models that are discussed in [141].

Tangling constraints make the consistent constraint management difficult, because all of the modifications need to be done on all occurrences of the constraints. Constraints appearing several times and crosscutting a transformation increase the time of constraint handling and the possibility of making an error during the modification. A few tangling constraints can make
the whole transformation scattered. The main idea is to construct AO constraints separately from the transformation rules, which encapsulate the crosscutting concerns, and provide a weaver method for the constraint propagation. In summary, the aim is to achieve a consistent constraint management (modification, deletion and propagation) with separating crosscutting constraints, and weaving them automatically.

5.3 Applying Aspect-Oriented Techniques for Constraint Management

This section gives an overview about the suggested aspect-oriented constraint management, and provides examples for the discussed methods. Section 5.3.1 introduces the concept of aspect-oriented constraints. Section 5.3.2 discusses the advantages of constraint aspects in visual model transformations. Section 5.3.2.1 presents an algorithm to normalize constraint aspects to achieve their (pure) canonical form, and finally, Section 5.3.2.2 deals with the steps of the constraint aspect creation from OCL constraints.
5.3.1 Aspect-Oriented Constraints

In Fig. 5.6, the presented OCL constraint \((\text{const}_{\text{age}})\) is propagated 6 times in a simple transformation that contains only two transformation rules. In model transformation, the OCL constraints assigned to PRNs represent the crosscutting concerns.

Aspect-oriented constraints are OCL constrains defined separately from the transformation and the transformation rules, and they are woven to the rules later, using a weaver method. In this approach, the context information of the aspect-oriented constraints is a type-based pointcut - it selects PRNs based on their metatype. The weaving process driven by the type-based pointcuts is referred to as \textit{type-based weaving}. In Fig. 5.6, \textit{Person} is the context of the constraint \(\text{const}_{\text{age}}\) and the target PRNs of the propagation are selected based on their metatype.

A \textit{weaving constraint} is similar to a property-based pointcut, it is also an OCL constraint that specifies the weaving but it is not propagated or used during the transformation. Weaving constraints facilitate to require any conditions during the weaving process. Therefore, this type of weaving is referred to as \textit{constraint-based weaving}. A weaving constraint can be used to represent one or many characters as a means of specifying more than one attribute during a search procedure. This enables to select multiple PRNs with a single specification. In the example presented in Fig. 5.6, we can use weaving constraints to refine the conditions expressed by AO constraints. The constraint \(\text{const}_{\text{age}}\) can be separated from the presented transformation rule, and using the following weaving constraint, it can be propagated to \textit{Person} type PRNs, whose names start with the '\text{'Employee'}' character sequence.

\[
\text{person.name} = '\text{'Employee'}'*'
\]

This means that the constraint \(\text{const}_{\text{age}}\) applied with the presented weaving constraint is not propagated to PRNs of the type \textit{Person} with name '\text{'ToEmploy'}'. The asterisk (*) is a wild-card that stands for any combination of letters. The AO constraint queries PRNs with the appropriate metatype, and the weaving constraint selects the appropriate PRNs from the result, based on their name. A \textit{weaving constraint} is a property-based pointcut. A weaving constraint can select any number of PRNs, based on their attribute values.

Having separated the constraints from PRNs, a weaver that facilitates the propagation of constraints to PRNs is also needed. Our approach solves the AO constraint propagation with the Global Constraint Weaver (GCW) algorithm. GCW algorithm is presented in Section 5.4.1.1.

This method facilitates the approach toward managing constraints using AO techniques. Similarly to aspects, the constraints are specified and stored independently of any model transformation rule or PRN, and they are linked to PRNs by the GCW.

The output of the weaver is not stored as a new transformation rule. The result is handled as a link between the constraints and a transformation rule. This link is referred to as \textit{weaving configuration}.

5.3.2 A New Type of Aspect: Constraint Aspect

VMTS is a visual approach: metamodels, models and transformations are defined by graphical models. But OCL constraints used to specify the transformations rules are textual expressions.
Therefore, the concept of *constraint aspect* has been developed. A constraint aspect expresses the same conditions as an OCL constraint in a visual way.

**Definition 5.1.** A *constraint aspect* is a pattern (structure) built from metamodel elements to which OCL constraints are assigned. A constraint aspect contains not only textual conditions described by the OCL constraints but structure, type and multiplicity conditions and weaving constraints as well.

The structure, type conditions and weaving constraints are checked at propagation time, while the OCL constraints are validated during the model transformation.

**Definition 5.2.** A *propagated constraint aspect* is a constraint aspect linked to the elements of a transformation rule. It forms a weaving configuration that contains the constraint aspect with the required conditions and the transformation rule.

Using OCL, starting from a specific object, we can navigate an association on a metamodel or a transformation rule, because transformation rules are built from metamodel elements to refer to other objects and their properties. To do so, we navigate the association by using the opposite association-end [91] [182].

```
context Person inv Const_P2:
   self.wife.isEmployed and
   self.wife.managedCompany.name = 'IDOS'
```

The value of this expression is a Set of objects on the other side of the `employee` association. If the multiplicity of the association-end has a maximum of one (0..1 or 1), then the value of this expression is an object.

During the execution, the constraint evaluation means the following. We must traverse the navigation path contained by the constraint, select the referred property of the object and check its value. The largest part of the computation complexity is the cost of the navigation. Therefore the aim is to use constraints and constraint aspects with as few navigation steps as it is possible.

**Definition 5.3.** The *canonical constraint form* of a constraint aspect is the form that contains the fewest possible navigation steps. The *pure canonical constraint* form of a constraint aspect is the form which does not contain any navigation steps.

A constraint aspect is a visual constraint. It fits better the visual transformation system than the OCL constraint. Fig. 5.7 introduces the concepts of the formulated definitions.

In Fig. 5.7a, a constraint aspect is depicted: `PersonA, PersonB, Company`, and the associations between them represent the structure of the constraint aspect. `Cons_P1` and `Cons_C1` are the OCL constraints propagated to the pattern of the constraint aspect. Fig. 5.7b represents a transformation rule with a propagated constraint aspect. The dashed line shows the propagation of the constraint aspect to LHS of the transformation rule.

In Fig. 5.7c/1, a constraint aspect is depicted with a propagated OCL constraint (`Cons_P2`). The dashed lines denote that `Cons_P2` refers to the PRNs `PersonB` and `Company`.

```
context Person inv Const_P2:
   self.wife.isEmployed and
   self.wife.managedCompany.name = 'IDOS'
```
Fig. 5.7. (a) Constraint Aspect, (b) Propagated Constraint Aspect, (c/1) Constraint Aspect, (c/2) Canonical Constraint Form, (c/3) Pure Canonical Constraint Form

Fig. 5.7c/2 shows the canonical constraint form of the constraint aspect depicted in Fig. 5.7c/1. The propagated OCL constraint is relocated to the PRN PersonB and the navigation steps of the constraint are updated. Therefore the constraint Cons_P3 refers to the PRN Company only. The constraint Cons_P3 contains the fewest possible navigation steps.

context Person inv Cons_P3:
    self.isEmployed and
    self.managedCompany.name = 'IDOS'

Fig. 5.7c/3 depicts the pure canonical constraint form of the constraint aspect (Fig. 5.7c/1). The constraint Cons_P3 is divided into two simpler constraints (Cons_P4 and Cons_C2) that are assigned to the PRNs PersonB and Company. Constraints Cons_P4 and Cons_C2 are on their appropriate place and do not contain navigation steps.

context Person inv Cons_P4:
    self.isEmployed

context Company inv Cons_C2:
    self.name = 'IDOS'
Fig. 5.8. A constraint aspect / left-hand side graph with propagated OCL constraints

The constraint aspect depicted in Fig. 5.7a is independently created from the transformation
rule (Fig. 5.7b), and it is propagated to the rule later by the constraint weaver.

5.3.2.1 Optimizing the Weaving Process with Normalization

OCL constraints often contain complex expressions with several navigation steps. The constraint
evaluation consists of two parts. (i) Selecting the object and its properties that the constraint
needs to be checked on, and (ii) executing the checking. In general, the larger part of the
evaluation is the first step, because of its computational complexity. Each navigation step in
a constraint means several queries on the model database. Therefore, it would be beneficial
to reduce the navigation steps contained by the constraints, because the eliminated navigation
steps accelerate the first part of the constraint evaluation.

This section provides a method with algorithms to normalize OCL constraints in UML class
diagram-based transformation rules and constraint aspects. This means eliminating as many
navigation steps from the constraints as possible. The method is also applicable for any UML
class diagram.

In Fig. 5.8, a sample LHS graph, an example for the problem, is depicted. The OCL
constraints propagated to LHS graph contain several navigation steps and built from subterms.
Subterms of the constraints are linked with and/or operations. The suggested normalization
methods, presented in next two subsections, apply constraint relocation and constraint decomposition techniques.

Relocating Constraints

Evaluating a constraint that contains navigation expressions requires more computational com-
plexity than a constraint without any navigation steps. To optimize the navigation, the OCL
constraints contained by a constraint aspect or a model can be normalized to obtain their canonical constraint form. Using the original OCL constraint without normalization, the navigation paths must be traversed at each evaluation of the OCL constraint during the execution. A normalized model incorporates the constraints on their most adequate places; therefore it includes the fewest possible navigation steps. Thus, we do not have to execute additional query operations during the constraint evaluation of the normalized model.

The first idea to reduce the number of the navigation steps contained by the constraints is to relocate the constraints from their original place to another. Relocation means that a constraint is removed from its original PRN and assigned to another such that the expression imposes the same constraint. The relocation algorithm must update the navigation paths contained by the relocated constraint, such that the constraint keeps its original conditions. For example if the constraint Cons_P2 in Fig. 5.8 is relocated from PRN PersonB to PRN PersonA, the navigation path self must be changed to self.wife and the path self.husband to self. The aim is to find a new place, a new PRN from where the constraint reaches all the necessary model elements with the fewest navigation steps.

Algorithm 5.1 presents the pseudo code of the NormalizeConstraint algorithm that generates the canonical form of a transformation rule, using constraint relocation techniques. The NormalizeConstraint algorithm can be applied for constraint aspects, transformation rules and UML class diagram-based models as well.

Algorithm 5.1 Pseudo code of the NormalizeConstraint algorithm

1: NormalizeConstraint (Model M)
2: for all PropagatedConstraint C in M that contains navigation do
3:   if C.HasCollectionOperation then
4:     continue
5:   if C.DestinationNodes.Count == 1 then
6:     Relocate C to DestinationNode of the C
7:   else
8:     minNumberOfSteps = CalculateNavigationSteps(DestinationNodes of the C, CurrentNode of the C)
9:     optimalNode = CurrentNode of the C
10:    for all Pattern Rule Node PRN in PatternRuleNodes of the M do
11:       numberOfSteps = CalculateNavigationSteps(DestinationNodes of the C, PRN)
12:       if numberOfSteps ≤ minNumberOfSteps then
13:          minNumberOfSteps = numberOfSteps
14:          optimalNode = PRN
15:     if (optimalNode != CurrentNode of the C and not IsZeroMultiplicityAllowed(CurrentNode of the C, optimalNode)) then
16:        UpdateNavigationPaths(C, optimalNode)
17:     Relocate(C, optimalNode)

The NormalizeConstraint algorithm processes all the OCL constraints propagated to the transformation rule. During constraint normalization, the algorithm modifies the navigation
Fig. 5.9. (a) Example left-hand sides with multiplicity 0..* and, (b) Example matched input model

paths. It calculates the paths between the original place of the constraint as well as the referred node (path $P_1$), and between the new place of the constraint and the referred node (path $P_2$). Finally, the algorithm replaces the path $P_1$ with the path $P_2$.

If zero multiplicity is allowed on the path $P$ that is selected to be replaced in constraint $C$, then the NORMALIZECONSTRAINT algorithm does not replace the path $P$. This means that the constraints are not moved along edges which allow the multiplicity 0. In Fig. 5.9a, a sample LHS graph is depicted with multiplicity 0..*. Fig. 5.9b shows an example for a matched input model. The constraint $C.P_1$ is assigned to PRN Person1 and refers to PRN Person2. If the constraint $C.P_1$ is relocated to Person2 then it is evaluated on the node $P2_a$. But if the constraint $C.P_1$ is on its original place, then it does not need to be evaluated on $P2_a$ because there is no edge between nodes $P1_a$ and $C1_a$. Consequently, moving a constraint along an edge which allows multiplicity 0 may cause that the constraint evaluation becomes incorrect. This is possible that (i) a constraint $C$ does not need to be evaluated on its original place but after the replacement it must be checked or (ii) the constraint $C$ needs to be evaluated on its original place but because of the replacement it is not checked.

The normalization is an offline algorithm: it works without any input model. In the case of complex multiplicities, like 1..*, it is not possible to predict how many instances are in the input model and in the matched submodel (a part of the input model) of the certain types. In the presented approach it is not necessary to traverse the navigation paths for each matched instance element. Each navigation path is parsed once, similarly to the case with multiplicity 1, and the constraint evaluation is performed on the collections of the PRNs. In each navigation step, the result can be a PRN collection not only a simple PRN. This collection is the input of the next navigation step, and the PRNs of this collection are evaluated without any extra navigation.

If not only multiplicity 1 is allowed, it is possible that after normalization the cost of the constraint evaluation increases. The reason for this is that a constraint navigating along 1..* edges can be more complex, than another OCL constraint that uses navigations along edges with 1 multiplicities. An example for this is depicted in Fig. 5.10. The number of the navigation steps is 3 (Person2 - Person1 - Company2 and Person2 - Company1) in the example LHS (Fig. 5.10a), and 2 (nCompany1 - nPerson4 - nCompany2) in the normalized LHS (Fig. 5.10b). But
the cost of the constraint evaluation on the example matched input model (Fig. 5.10c) is higher in case of the normalized version, because of the allowed 1..* multiplicities.

As far as complex multiplicities are concerned, the optimization depends on the actual input model. As it has been mentioned, the normalization is an offline algorithm, it is not possible to state further normalization solution for the normalized models which contain complex multiplicities.

In general, a constraint can be relocated if and only if the constraint expresses the same conditions both before and after the relocation. Therefore, in case of the complex multiplicities (e.g. 1..*) the normalization has to take into consideration the constraint types. There are constraints that cannot express the same conditions after constraint relocation. An example for it is a constraint that contains a count operation. In this case it is not possible to relocate the constraint to a node, which originally was the target of the count operation. Similarly, the normalization should be careful with the collection operations (exists, forAll, select, collect, and iterate) and collection-related type operations (e.g. size, includes, excludes, count, includesAll, excludesAll, isEmpty, notEmpty, and sum) executed on the resulted Set of a navigation.

The computational complexity of the NormalizeConstraint algorithm is $O\left(\sum_{i=1}^{c} (n_i + v^3)\right)$, where $c$ denotes the number of the propagated constraints contained by the transformation rule, $n_i$ is the number of the navigation steps contained by the constraint $i$, and $v$ denotes the number of PRNs in the transformation rule. The complexity of finding the shortest path in a transformation rule is $v^2$. It must be executed $v$ times for each PRN in constraint aspect).
The NormalizeConstraint algorithm is applicable for paths without zero multiplicities and for constraints without collection or collection-related type operations executed on Sets resulted by navigations. The algorithm results that the number of the navigation steps in resulted constraints is as few as possible that can be achieved with constraint relocation.

**Proposition 5.4.** Let $H$ be an optional input model that do not contain zero multiplicities and constraints with collection or collection-related type operations executed on the results of the navigations. NormalizeConstraint algorithm using constraint relocation eliminates all possible navigation steps in the constraint expressions of the model $H$. The constraints in the resulted model $H'$ implement the original conditions and contain as few navigation steps as it is possible.

**Proof.** Appendix A.

The goal of the constraint normalization is to achieve the pure canonical form, which does not contain navigation steps. Using NormalizeConstraint algorithm, it is not possible in all cases, because a constraint is often built from subterms and linked with boolean operators (for example `self.wife.managedCompany.name = 'IDOS'` or `self.managedCompany.name = 'IDOS'`), or requires property values from different nodes (`self.age + self.husband.age > 60` - see Fig. 5.8), therefore the algorithm cannot eliminate all the navigation steps.

**Decomposing Constraints**

To achieve the pure canonical form of a constraint aspect or model the concept of the AND/OR Clauses have been developed.

**Definition 5.5.** A clause contains any number of OCL expressions, along with any number of other clauses and exactly one operation (and/or/xor) between them.

Separating the clauses, the number of the navigation steps contained by the OCL expression, and the complexity of the constraint evaluation during the constraint validation process can be reduced. It is simpler to evaluate the and/or/xor operations between the members of a clause, than to walk through the navigation paths contained by the constraints.

The concept of the constraint decomposition is the following. If the subterms of a constraint are connected only with and operations, it is enough to create new constraint for each subterm, set the context information for them, propagate this new constraint to the model, and finally remove the original constraint from the model. For example the constraint $Cons_C1$ in Fig. 5.8 contains two subterms connected with and operation. $Cons_C1$ can be decomposed into two separate constraints: $Cons_C1a$ and $Cons_C1b$ (Fig. 5.11). These modifications do not change the result of the constraint evaluation, because the original terms are connected with and operation and the whole expression is true only if all of the subterms are true. On the modified model - with the new constraints - the situation is the same, the evaluation of the constraints contained by the model is true if and only if all of the constraints return true.

For constraints that contains subterms linked with or/xor operations the mentioned method is not applicable. During the evaluation of such a constraint, it is not required that all the
Fig. 5.11. The example constraint aspect / LHS with normalized OCL constraints

subterms are true. Therefore, it would be unnecessarily strict condition to decompose the constraint and propagate the subparts to the model. In the original constraint it is enough that only one of the subterms is true, then the value of the whole constraint is true because of the or operation. In the case of xor operation the required condition is that at least one of the constraint is true and one false. Contrarily, in the case of separately propagated subterms, all condition must be true to make the whole expression true. To solve this problem AND/OR clauses-based constraint management is applied. For example, the constraint Cons in Fig. 5.8 is decomposed into two constraints (Cons.Clause1a and Cons.Clause1b) which are contained by the OR clause Clause1 (Fig. 5.11). The constraints do not contain navigation steps that accelerate the evaluation, but an additional reference is maintained in order to check the two clauses together. This means that it is enough that only one of the clauses is true.

Algorithm 5.2 and 5.3 show the pseudo code of the DecomposeConstraint and CreateClause algorithms that generate the pure canonical form of the transformation rule.

A model or a constraint aspect is passed to the algorithm DecomposeConstraint. The algorithm processes the OCL constraints propagated to the model (the main for all loop). If the actual constraint requires property values from different nodes, the algorithm calls the algorithm NormalizeConstraint to reduce the number of the navigation steps. Otherwise, the algorithm examines if the constraint contains or/xor operations or only and operations. If the subterms of a constraint are connected with and operations, new constraints are created for each subterm (DecomposeConstraint - lines 3-11).

Constraint that contains subterms linked with or/xor operations is processed with the CreateClause method. The parentheses could change the order of the operations. Therefore, if the input term contains parentheses, then the CreateClause method identifies the subterms based on them (lines 4-5) otherwise based on the or/xor operations (line 7). The method creates a new
Algorithm 5.2 Pseudo code of the DecomposeConstraint algorithm

1: DecomposeConstraint (Model $M$)
2: for all PropagatedConstraint $C$ in $M$ that contains navigation do
3: if ContainsCalculatedExpression($C$) then
4: NormalizeConstraint($M$, $C$)
5: else
6: if $C$ does not contain or/xor operation then
7: for all term in ConstraintTerms of the $C$ do
8: newContraint = CreateConstraint(term)
9: SetContext(newContraint)
10: PropagateConstraint($M$, newContraint)
11: RemoveConstraint($M$, $C$)
12: else
13: clause = CreateClause($C$)
14: RemoveConstraint($M$, $C$)
15: PropagateClause($M$, clause)

Algorithm 5.3 Pseudo code of the CreateClause algorithm

1: CreateClause (ContraintTerm $Term$) : Clause
2: if $Term$ does not contain subterms then
3: return $Term$
4: if $Term$ contains parentheses then
5: subTerms = GetSubtermsByParentheses($Term$)
6: else
7: subTerms = GetSubtermsByOrXorOperation($Term$)
8: newClause = CreateClause(operation between subTerm)
9: for all actualTerm in subTerms do
10: subClause = CreateClause(actualTerms)
11: AddSubclauseToClause(newClause, subClause)
12: return newClause

clause with the appropriate operation (and, or, xor - newClause - line 8), recursively creates the subclauses, adds them to the newClause (line 9-11), and finally returns the newClause.

In Fig. 5.11, the result of the normalization process for the model in Fig. 5.8 is depicted. As it is presented, the Cons__C1 is decomposed into Cons__C1a and Cons__C1b. A clause is created from Cons__P1. Unfortunately, the presented algorithms do not eliminate the navigation if the constraint requires property values from different nodes (Cons__P2).

Proposition 5.6. Applying the algorithm DecomposeConstraint the number of the navigation steps in the constraints contained by the output model is few as possible that can be achieved with constraint relocation and constraint decomposition.

Proof. Appendix A. □
Proposition 5.7. Applying NormalizeConstraint and DecomposeConstraint algorithms for any UML class diagram-based input model does not modify the result of the constraint evaluation.

Proof. Appendix A.

In summary, the presented normalization methods can be used to preprocess the model transformation rules. They must be accomplished once for a transformation rule, and the result, the normalized rule, can be executed any number of times with more efficient constraint evaluation.

The main limitation of the normalization algorithm is the following. If zero multiplicity is allowed on the path $P$ that is selected to be replaced in constraint $C$, then the NormalizeConstraint algorithm does not replace it. Constraints are not moved along edges which allow zero multiplicity, because it may cause that the constraint validation becomes incorrect. Furthermore, the method does not support the normalization of the constraints with collection or collection-related type operations executed on Sets resulted by navigations.

5.3.2.2 Creating Constraint Aspect from OCL Constraint

Fig. 5.12 introduces the creation process of constraint aspect from an OCL constraint along with the normalization of the created constraint aspect. The lines with numbers from 1 to 4 show the steps of the constraint aspect creation: the algorithm identifies the context type ($Person$) and the referred types by the association ends ($Company$, $Person$). Based on the types, it builds the pattern. Finally, the algorithm assigns the OCL constraint to the root PRN of the created constraint aspect. In Fig. 5.12b, the dashed lines denote that the constraint Cons_P1 contains path expressions. The lines 5 and 6 show the constraint normalization (decomposition and relocation). The pseudo code of the algorithm CreateConstraintAspect is as follows.

Algorithm 5.4 Pseudo code of the CreateConstraintAspect algorithm

1: CreateConstraintAspect (VMTSConstraint $C$): VMTSConstraintAspect
2: $CA = \text{CreateCA}(C.\text{Context})$
3: for all NavigationStep $N$ in $C$ do
4: $PRN = \text{CreatePRN}(\text{.TypeOf}(N.\text{DestinationNode}))$
5: LinkPRN($CA$, $PRN$)
6: PropagateConstraint($CA$, $C$)
7: return $CA$

The computational complexity of the CreateConstraintAspect method is $O(n)$, where $n$ is the number of the navigation steps contained by the processed constraint.

Informally, if the constraint aspect $CA$ is created from an OCL constraint $C$ using the CreateConstraintAspect algorithm, and the normalized constraint aspect $CA'$ is created from the $CA$ with NormalizeConstraint algorithm, then $C$, $CA$ and $CA'$ are equivalent in the sense of the contained conditions. Furthermore, after their propagation to a transformation the constrained transformation is also equivalent during the transformation process.
**Proposition 5.8.** The constraint aspect $CA$ is created with the `CreateConstraintAspect` algorithm from the OCL constraint $C$. $S_1$ and $S_2$ are two identical transformation rules, $CA$ is propagated to $S_1$, and $C$ is propagated to $S_2$. Then no input model $H$ exists for which transformation rules $S_1$ and $S_2$ produce different result models.

**Proof.** Appendix A.

**Proposition 5.9.** The normalized constraint aspect $CA'$ is created with the `NormalizeConstraint` algorithm from the $CA$ constraint aspect. $S_1$ and $S_2$ are two identical transformation rules, $CA'$ is propagated to $S_1$, and $C$ is propagated to $S_2$. Then no input model $H$ exists for which transformation rules $S_1$ and $S_2$ produce different result models.

**Proof.** Appendix A.

**Proposition 5.10.** The normalized constraint aspect $CA'$ is created from the OCL constraint $C$ with the `CreateConstraintAspect` and `NormalizeConstraint` algorithms. $S_1$ and $S_2$ are two identical transformation rules, $CA'$ is propagated to $S_1$, and $C$ is propagated to $S_2$. Based on Proposition 5.8 and 5.8, no input model $H$ exists for which the transformation rules $S_1$ and $S_2$ produce different result models.

**Proof.** Appendix A.

### 5.4 Weaver Algorithms

The weaving method is an offline algorithm, it should be accomplished once for a set of constraints and a transformation. Weaving consists of an optimization part, presented in section 5.3.2.1, and constraint assignment. This section provides the algorithms that have been developed for AO constraint and constraint aspect assignment.
5.4.1 Constraint Weaving

Two weaver algorithms have been developed: the Global Constraint Weaver (GCW) and the Constraint Aspect Weaver (CAW). They use similar methods for constraint propagation. The difference between them is that GCW weaves AO constraints, while CAW propagates constraint aspects to model transformation rules.

In Fig. 5.13, the input and the output of the weavers are depicted. The input of the GCW is the transformation rule, the aspect-oriented constraints, and the weaving constraints. The input of the CAW is the transformation rule and the constraint aspects. The output of both weavers is the constrained transformation rule.
5.4.1.1 The Global Constraint Weaver Algorithm

The GCW algorithm is passed a transformation with any number of transformation rules and a constraint list (AO constraints and weaving constraints). The algorithm propagates the AO constraints to the appropriate PRNs contained by the transformation rules.

GCW algorithm has three main steps to select PRNs to which the constraint must be linked.

(i) It obtains PRNs from the transformation rules with a metatype corresponding to the context information of the AO constraint. (ii) If the constraint contains navigation steps (for example `self.wife.managedCompany.name = 'IDOS'` in Fig 5.8), then the algorithm checks the structure of the transformation rule for each previously selected PRN. It is validated whether it satisfies the pattern required by the navigation paths. The pattern of the transformation rule is suitable for a constraint if starting from PRN to which the constraint is linked, one can traverse the paths described by the navigation paths of the constraint. In other words, the pattern of the transformation rule contains PRNs with an appropriate type, and relation between them which facilitates to traverse the navigation paths contained by the constraint. (iii) In the third step, examining the transformation rules, the algorithm decides if it is necessary to assign a constraint to each transformation rule, or it is sufficient to assign it only to the first rule as a precondition, and to the last rule as a postcondition. If an intermediate state modifies one of the properties contained by the constraint, the algorithm assigns the constraint to this intermediate state to prevent a violated condition not being revealed until the end of the transformation.

The pseudo code of the GCW algorithm is as follows.

**Algorithm 5.5** Pseudo code of the GlobalConstraintWeaver algorithm

```plaintext
1: GlobalConstraintWeaver (VMTSConstraint[] Cs, VMTSConstraint[] WeavingCs, VMTSTransformation T)
2: for all VMTSConstraint C in Cs do
3:   for all TransformationRule S in T do
4:     nodesWithProperMetaType = MetaTypeBasedSearching(ContextInfo of C, S)
5:     nodesToCheck = EvaluateWeavingConstraints(nodesWithProperMetaType, WeavingCs)
6:     if IsRequiredToWeave(C, nodesToCheck, out nodesToWeave) then
7:       WeaveConstraint(C, nodesToWeave)
```

The following proposition states that the Global Constraint Weaver propagates constraints only to the necessary places, and optimizes the constraint validation for the whole transformation process.

**Proposition 5.11.** \( T_1 \) and \( T_2 \) are two identical transformations which contain any number of transformation rules, and \( C \) is an OCL constraint. (i) \( C \) is propagated to \( T_1 \) using Global Constraint Weaver algorithm. (ii) Based on the required transformation type (validation, preservation or guarantee), \( C \) is enlisted in all transformation rules of \( T_2 \). Then transformations \( T_1 \) and \( T_2 \) produce the same model as a result step by step.

**Proof.** Appendix A.
Corollary 5.12. \( T_1 \) and \( T_2 \) transformations step by step produce the same result model, therefore, the whole transformations \( T_1 \) and \( T_2 \) have the same effect in all cases during the transformation process.

Remark 5.13. In the VMTS approach, transformations are executed in transactions. This means that if a transformation is unsuccessful, the changes made on the input model are rolled back, therefore, it results the original state of the model before the transformation. Consequently, if transformation \( T_1 \) fails in the first step and transformation \( T_2 \) fails in the step \( n \), then because of the rollback at the end the transformations \( T_1 \) and \( T_2 \) have the same effect.

The computational complexity of the \textit{GLOBALCONSTRAINTWEAVER} algorithm is at most \( O(\sum_{i=1}^c \sum_{j=1}^s (\log v_r + n_{ij} + v_r^h + n_{ij})) \), where \( v_r \) denotes the number of the nodes in the actual transformation and \( \log v_r \) is the complexity of querying a node from the database. The variable \( c \) denotes the number of the constraints to be propagated to the transformation rules, \( s \) is the number of the transformation rules in the transformation. Moreover, \( n_{ij} \) is the number of the nodes in the transformation rule \( j \) with the same type as the context of the constraint \( i \). In this case \( v_r^h \) is the worst case for finding an isomorphic submodel [171], where \( h \) is the size of the submodel.

Detailed computational complexity considerations about constraint evaluation and navigation can be found in Appendix B.

5.4.1.2 The Constraint Aspect Weaver Algorithm

The CAW algorithm, using similar methods to GCW, weaves constraint aspects to model transformation rules.

\textbf{Algorithm 5.6} Pseudo code of the \textit{CONSTRAINTASPECTWEAVER} algorithm

\begin{verbatim}
1: CONSTRAINTASPECTWEAVER (VMTSConstraintAspect[] CAs, VMTSTransformation T)
2:   for all VMTSConstraintAspect CA in CAs do
3:     for all Transformation Rule S in T do
4:       matches = METATYPEBASEDMATCHING(pattern of the CA, S)
5:       matches = EVALUATEWEAVINGCONSTRAINTS(matches)
6:       for all VMTSConstraint C in CA do
7:         nodesToCheck = GETNODESTYPE(C, matches)
8:         if ISREQUIREDTOWEAVE(C, nodesToCheck, out nodesToWeave) then
9:           WEAVECONSTRAINTASPECT(CA, nodesToWeave)
10:          break
\end{verbatim}

The CAW algorithm checks each transformation rule individually. In each rule, it searches for an isomorphic submodel to the structure of \( CA \) (line 4) and checks the found structures based on the weaving constraints. For each constraint \((CA_{\dots}C_1 \ldots CA_{C_n})\) contained by \( CA \), the algorithm selects PRNs from the matches with the metatype that corresponds to the context information of the actual constraint (lines 6-7). CAW uses the \textit{ISREQUIREDTOWEAVE} method to decide if a transformation rule requires \( CA \) to be propagated (line 8). If at least one of
the constraints \((CA_{C_1} \ldots CA_{C_n})\) contained by CA needs to be propagated, then the whole constraint aspect is linked (line 9).

The computational complexity of the \texttt{ConstraintAspectWeaver} method is at most \(O(\sum_{i=1}^{ca} \sum_{j=1}^{s} (n_j^{k_i} + \sum_{p=1}^{c_i} m_{jp}))\), where \(ca\) denotes the number of the constraint aspects, \(s\) is the number of the transformation rules, \(c_i\) is the number of the constraints contained by the constraint aspect \(i\), \(n_j^{k_i}\) is the complexity of the metatype-based matching (worst case), \(n_j\) is the number of PRNs contained by the transformation rule \(j\) and \(k_i\) is the number of PRNs contained by the constraint aspect \(i\). The \(m_{jp}\) is the number of PRNs selected by the \texttt{GetNodesByType} method.

5.4.1.3 Comparison of the Weaver Algorithms

Both OCL expressions and constraint aspects can contain a structural representation. Therefore, specifying a constraint in the structure and with OCL navigation means two equivalent ways. This equivalence is preserved by the two weaver algorithms. If the same path is specified in OCL and with a structure, the two algorithms provide the same result.

Informally, assume that an AO constraint \(C\) and a constraint aspect \(CA\) express the same conditions, if \(C\) is propagated to a transformation with the \texttt{GCW} and \(CA\) with the \texttt{CAW} to the same transformation, then the results of the two constraint propagation (the constrained transformations) are equivalent.

\textbf{Proposition 5.14.} The normalized constraint aspect \(CA'\) is created from the with OCL constraint \(C\) with the \texttt{CreateConstraintAspect} and \texttt{DecomposeConstraint} algorithms. \(T_1\) and \(T_2\) are two identical transformations, which contain any number of transformation rules. \(C\) is propagated to \(T_1\) using the Global Constraint Weaver algorithm and \(CA'\) is propagated to \(T_2\) using the Constraint Aspect Weaver algorithm. Then for any finite input model \(H\) the transformations \(T_1\) and \(T_2\) produce the same result models step by step.

\textit{Proof.} Appendix A.

However, it is more efficient to work with constraint aspects than OCL constraints, because during propagation of the constraint aspects the metatype-based searching can be used for pattern matching [146] to reduce the possible places of the constraint assignment. Then the \texttt{IsRequiredToWeave} algorithm has to be executed only on the selected places to decide if it is necessary to assign the given constraint aspect. In case of OCL constraints, the \texttt{GCW} algorithm uses the \texttt{CheckStructure} method to find the structurally appropriate places in the transformation rules, while the complexity of the metatype-based searching is equal to the complexity of finding isomorphic submodel only in the worst case.

The result of the normalization is that the constraints are on their optimal place. The normalization of the OCL constraints can be achieved only after their propagation when they are linked to the PRNs. There are two problems related to it: (i) the result of the weaving is a configuration, where constraints are assigned virtually to PRNs that makes difficult the execution of the normalization algorithm. (ii) On the other hand, the normalization should be executed on all places the constraint is propagated to. This means that the same operation should be
achieved several times. Contrary, the normalization of a constraint aspect can be executed before the weaving and should be performed only once. Therefore, the normalization is performed only for the constraint aspects, thus during the transformation process it is more efficient to work with constraints that contain propagated constraint aspects instead of linked OCL constraints. Consequently, constraint aspects accelerate both the weaving and the transformation processes.

5.5 Overview of Different Aspect-Oriented Constraint Notations

An overview of the different aspect-oriented constraint notions and the presented algorithms is depicted in Fig. 5.14.

Creating and normalizing constraint aspects need to be performed once for a constraint. The whole weaving process must be executed once for a transformation. This obviously takes time, but all of they are offline algorithms, preceding the execution time. After they have been accomplished once, their results the constrained transformations, can be reused for any number of models.

Normalized constraint aspects contain the constraints on their adequate places, with as few navigation steps as possible. Therefore, for the transformation rules that allow only multiplicity 1 the following proposition can be stated.

**Proposition 5.15.** The normalized constraint aspect $CA'$ is created from the OCL constraint $C$ with the `CREATECONSTRAINTASPECT` and `DECOMPOSECONSTRAINT` algorithms. The evaluation cost of the propagated normalized constraint aspect $CA'$ reaches the evaluation cost of the propagated OCL constraint $C$ in the worst case only.

**Proof.** The normalized constraint aspect $CA'$ contains as many navigation steps as the OCL constraint $C$ in the worst case only.
5.6 Example for Aspect-Oriented Constraint Management

To show the practical relevance of the presented approach the constraint management part of the case study Class2RDBMS that generates database model from class diagram is provided in this section.

The transformation rule ProcessAssociation of the transformation Class2RDBMS is depicted Fig. 5.15. This transformation rule processes the associations of the input class diagrams. It creates distinct tables for many-to-many (N:N) associations with the appropriate columns. Furthermore, in case of the one-to-many (1:N) and one-to-one (1:1) associations this rule merges the association related columns into the existing tables. It is relevant to process only the associations that link non-abstract classes, which requires to propagate constraints to the PRNs. To ensure this condition, the constraint that requires a class to be non-abstract should be propagated to this transformation four times.

The control flow model of the transformation is presented in Fig. 6.1. The model contains nine transformation rules and six different constraints that specify properly the execution of the transformations. Two of the constraints crosscut the whole transformation. The first one that requires a class to be non-abstract (NonAbstract) appears 30 times in the transformation, the second one that requires a class to be abstract is applied 16 times. Furthermore, in Section 2.4 emerged the demand that transformation rules creating and modifying tables should guarantee that foreign keys do not allow NULL values.

context Class inv NonAbstract:
not self.abstract

context Table inv PrimaryAndForeignKey:
not self.columns->exists(c | (c.is_primary_key or c.is_foreign_key) and c.allows_null)

The problem of the crosscutting constraint and constraint propagation have been solved with aspect-oriented techniques described in this chapter. The constraint NonAbstract, propagated to the transformation rules of the transformation Class2RDBMS, validates that in the resultant database model only the non-abstract classes have generated table (Section 2.4). The constraints NonAbstract and PrimaryAndForeignKey are propagated to the transformation with Global Constraint Weaver algorithm (Section 5.4.1.1) using validate and guarantee type constraints. The constraint assignment can be solved with Constraint Aspect Weaver algorithm (5.4.1.2) as well. The weaving configurations should be defined once and the weaving is achieved automatically that assigns the constraints to several palaces. In Fig. 5.16 is depicted the propagation result of the constraint NonAbstract.

5.7 Chapter Summary

A disadvantage of our earlier metamodel-based model transformation approach [131] can be seen in many tangling constraints throughout of the transformation rules. In metamodel-based model transformation, the two main advantages of the aspect-oriented constraint management
Fig. 5.15. Transformation rule \textit{ProcessAssociation}

Fig. 5.16. The result of the constraint propagation with Global Constraint Weaver

are the following. (i) It eliminates the crosscutting constraints from model transformations. (ii) Using AO methods, constraints become aspects. This means that the transformation rules can be executed with or without the propagated constraints as well. Moreover, the optimized transformation rules and constraints can be reused. Hence, the transformation can be executed with different propagated constraint set based on the required conditions. Furthermore, constraints are defined and stored independently from the transformations. Therefore they can be propagated to different transformations, thus, the constraint themselves can also be reused.

The problem of crosscutting constraints in metamodel-based model transformation rules, and the aspect-oriented constraint management methods have been discussed. The concept of AO constraints, weaving constraints, type-based weaving, constraint-based weaving and a new type of aspect, the constraint aspect have been introduced. In addition, the algorithms creating
and normalizing the constraint aspects have been presented. In the scope of the constraint normalization, an overview has been given about the benefits of OCL constraint relocation and decomposition. The presented normalization methods can be applied not only to constraint aspects and transformation rules but also for any UML class diagram-based models [109] [125]. Finally, two weaver methods have been also provided for aspect-oriented constraint and constraint aspect propagation.

Using aspect-oriented constraint management in visual model transformation, it has been observed that the maintainability and understandability of the transformation rules have been increased along with the attached constraints. With the help of this technique, several benefits have been achieved. Consistent constraint modification and simple constraint removal have become possible. The same constraint does not appear repetitiously in many different places. Moreover, it is not necessary for the transformation rules to be aware of the constraints, or for the modeler who creates the transformation rules. The provided weaver algorithms work on whole transformation, they handle all transformation rules contained by the transformation. Therefore, the result of the weaving, the constrained transformation as a whole satisfies the conditions required by the constraints [109]. The discussed methods have successfully been applied in industrial projects [120] [123] [130] [141].

Working with constraints aspects is more efficient than OCL constraints. The normalization must be accomplished once and not at each constraint evaluation. During the propagation the structure of the constraint aspects facilitates the more efficient weaving process. Therefore, constraint aspects make the weaving and the transformation processes more efficient.

The introduced approach can be generalized to other transformation languages which facilitate to assign constraints to transformation rules. The presented concepts and algorithms can be reused with minor, approach-related modifications.

The main limitation of the aspect-oriented constraint management is that it requires more preprocessing steps than the general approach. Constraints and transformation rules have to be defined separately, and then the propagation of the constraints to the transformation rules must be performed automatically [109].
Chapter 6

Control Flow Related Properties of Model Transformations

Visual model transformations often need to follow an algorithm that requires a strict control over the execution sequence of the transformation rules. Therefore, in Visual Model Processors (VMPs) the execution order of the transformation rules is crucial. This chapter presents a visual control flow language that facilitates composing complex model transformations of simple transformation rules and executing them. The VMTS Visual Control Flow Language (VCFL) uses stereotyped UML activity diagrams to specify control flow structures and OCL constraints to choose between different control flow branches. This chapter introduces VCFL, discusses its termination properties, provides an algorithm to support the termination analysis of VCFL transformations and presents a transformation rule composition method.

The motivation of the work presented in this chapter is to support the control flow in visual model transformation systems and to give a method that can be used to predict the behavior of the transformations, for example their termination properties. In general, the termination is undecidable [166], but there are certain cases when it can be proved.

Section 6.2 introduces the principles of the VCFL. Section 6.3 provides algorithms to check whether a transformation contains isolated or illegal transformation rules and to validate that the OCL constraints assigned to a decision object are disjoint. Furthermore, Section 6.4 gives considerations and algorithms to compose metamodel-based model transformation rules that facilitates to predict transformation properties. Such a predictable property is the termination of the transformations. Section 6.5 contains considerations related to the termination properties of VCFL and provides an algorithm to support the termination analysis of transformations. The suggested algorithm is an offline algorithm, as an input it uses only the control flow model to make the decision. This means that the decision is independent from any input model the transformation is applied to. Finally, a summary is given about the discussed results.
Chapter 6. Control Flow Related Properties of Model Transformations

6.1 Backgrounds and Related Work

In Chapter 7 a synopsis is given on model transformation systems in order to compare their control flow support. These approaches are also included in the related work of the VMTS visual control flow language.

Since the formalism and the results of the DPO approach are used, a summary of the necessary definitions and results are given, based on [54].

**Definition 6.1.** A production \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) consists of finite graphs \( L, K \) and \( R \), called left-hand side, gluing graph and right-hand side respectively, and two injective graph morphisms \( l \) and \( r \).

For practical purposes, it is important to restrict the applicability of a production by application conditions. In particular, negative application conditions can be used, which forbid the existence of a certain subgraph.

**Definition 6.2.** A negative application condition of a production \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) is of the form \( \text{NAC}(x) \), where \( x : L \to X \) is an injective graph morphism. A graph morphism \( m : L \to G \) satisfies \( \text{NAC}(x) \) if there does not exist an injective graph morphism \( p : X \to G \) with \( p \circ x = m \).

\[
\begin{array}{cccc}
X & \xleftarrow{s} & L & \xrightarrow{r} & K & \xrightarrow{r} & R \\
p & \downarrow & id & \downarrow & G
\end{array}
\]

**Definition 6.3.** Given a graph production \( p = (L \xleftarrow{l} K \xrightarrow{r} R) \) and a graph \( G \) with a graph morphism \( m : L \to G \), called match. If \( m \) satisfies all negative application conditions of \( p \), a direct graph transformation \( G \xrightarrow{p,m} H \) from \( G \) to a graph \( H \) is given by the following double pushout (DPO) diagram, where (1) and (2) are pushouts.

\[
\begin{array}{cccc}
X & \xleftarrow{s} & L & \xrightarrow{r} & K & \xrightarrow{r} & R \\
p & \downarrow & id & \downarrow & G & \xrightarrow{D} & H
\end{array}
\]

A sequence \( G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n \) of direct graph transformations is called a graph transformation and is denoted as \( G_0 \xrightarrow{*} G_n \). For \( n = 0 \) we have the identical graph transformation \( G_0 \xrightarrow{id} G_0 \).

We say \( p \) is applicable to \( G \) via \( m \), if \( m \) satisfies the NACs of \( p \), pushouts (1) and (2) exist, and the resulting graph \( H \) satisfies additional constraints given by the system. In this thesis we assume injective matches \( m \) and comatches \( n \).

**Definition 6.4.** A graph transformation system \( GTS = (P) \) consists of a set of graph productions \( P \) with or without negative application conditions. For a graph transformation system, there may be given a set of finite input graphs.
The definition for the termination of a graph transformation system is taken from [166].

**Definition 6.5.** A graph transformation system \( GTS = (P) \) terminates if there is no infinite sequence of direct graph transformations \( G_0 \Rightarrow G_1 \Rightarrow \ldots \) applying rules from \( P \) starting from any input graph \( G_0 \).

**Definition 6.6.** Given two productions \( p_1 = (L_1 \xleftarrow{e_1} K_1 \rightarrow R_1) \) and \( p_2 = (L_2 \xleftarrow{e_2} K_2 \rightarrow R_2) \), 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. The \( E \)-concurrent production \( p_1 \ast_E p_2 \) is a production \( p = (L \xleftarrow{e} K \rightarrow R) \) 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 \( n_2 \).

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

A transformation \( G \xrightarrow{p_1,m_1} H \xrightarrow{p_2,m_2} G' \) is called \( E \)-related to \( p_1 \ast_E p_2 \) 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.

Using the definition above, an algorithm can be constructed to compute \( E \)-concurrent productions.

**Algorithm 6.1** 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 6.7 (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.

6.1.1 Termination Criteria for Contextual Layered Graph Grammars

Contextual layered graph grammars (CLGGs) have been used in parsing, as they provide a natural way to steer the parsing process, thereby reducing its non-determinism and its complexity. A contextual layered graph grammar is a construct $CLGG = (G, T, RULES, cl, dl, rl)$, where $G$ is a labeled graph, called the initial graph, $T$ is a set of node and edge types of labels and $RULES$ is a set of rules. The layering functions $cl$, $dl$, and $rl$ assign a creation and a deletion layer to elements of $T$ and a unique layer to each rule $S \in RULES$, respectively. In [53], the following concrete termination criterion for CLGGs was discussed.

The criteria for nondeleting rules are based on the single or double pushout approach [33]. Moreover negative application conditions (NACs) are used given by an injective morphism $n : L \rightarrow N$. The match $m : L \rightarrow G$ satisfies the NAC if there is no injective morphism $q : N \rightarrow G$ with $m = q \circ n$. Labels $LABELS$ are defined in the traditional way by label sets or in correspondence with the metamodel.

The deletion layer conditions express that the last creation of a node with a certain label should precede the first deletion of a node with the same label. On the other hand, nondeletion layer conditions ensure that if an element of label $l$ occurs in the LHS of a rule then all elements of the same label were already created in previous layers.

Definition 6.8. Layered Graph Grammar. A graph grammar with rules $RULES$ and labels $LABELS$ is called layered graph grammar if for each rule $S \in RULES$ there is a rule layer $rl(S) = k$ with $0 \leq k \leq k_0$ ($k, k_0 \in \mathbb{N}$) where $k_0$ is the number of layers. Moreover, for each label $l \in LABELS$ there is a creation and a deletion layer $cl(l), dl(l) \in \mathbb{N}$ and each layer $k$ is either a deletion layer or a nondeletion layer satisfying the following conditions for all $S \in RULES$ with $rl(S) = k$:
If $k$ is a deletion layer

**Deletion Layer Conditions**

1. $S$ is deleting at least one item
2. $0 = cl(l) = dl(l) = k_0$ for all $l \in \text{LABELS}$
3. $S$ deletes $l \Rightarrow dl(l) \leq rl(S)$
4. $S$ creates $l \Rightarrow cl(l) > rl(S)$

If $k$ is a nondeletion layer

**Nondeletion Layer Conditions**

1. $S$ is nondeleting, i.e. $K = L$ s.t. $S : L \rightarrow R$
2. $S$ has NAC $n : L \rightarrow N$ and there is an injective $n' : N \rightarrow R$ with $n' \circ n = S$
3. $x \in L \Rightarrow cl(\text{label}(x)) = rl(S)$
4. $S$ creates $l \Rightarrow cl(l) > rl(S)$

**Definition 6.9. Termination of Layered Graph Grammars.** A layered graph grammar with finite start graph $G_0$ and rules $\text{RULES}$ terminates, if there is no infinite derivation sequence from $G_0$ via $\text{RULES} = [\text{RULES}_k = \{S \in \text{RULES} | rl(S) = k\}]_{k=0}^{k_0}$, where starting with layer $k = 0$ rules $S \in \text{RULES}_k$ are applied as long as possible before going over to layer $k + 1 \leq k_0$.

**Lemma 6.10. Termination of Layered Graph Grammars with Deletion.** Each layered graph grammar with deletion terminates.

**Definition 6.11. Transformation and Essential Match.** Given a nondeleting graph grammar with injective matches a nondeleting rule $S$ is given by an injective morphism $r : L \rightarrow R$, and a match $m : L \rightarrow G$ is an injective morphism leading to a transformation rule $G \Rightarrow H$ via $(r,m)$ defined by the pushout (1) of $r$ and $m$, where $d : G \rightarrow H$ is called *tracking morphism* of $G \Rightarrow H$ via $(r,m)$.

\[
\begin{array}{c}
L \xrightarrow{r} R \\
\downarrow \quad \downarrow (1) \\
G \xrightarrow{d} H
\end{array}
\]

Since $r$ and $m$ are injective morphisms, pushout properties (1) imply that also $d$ and $m^*$ are injective. Given a transformation $G_0 \Rightarrow H_1$ i.e. a sequence of transformation rules with induced injective tracking morphism $d_1 : G_0 \rightarrow H_1$ a match $m_1 : L \rightarrow H_1$ of $L$ in $H_1$ has an essential match $m_0 : L \rightarrow G_0$ of $L$ in $G_0$ if we have $d_1 \circ m_0 = m_1$. There is at most one essential match $m_0$ for $m_1$, because $d_1$ is injective.

**Lemma 6.12. In each derivation sequence starting from $G_0$ of a nondeleting layered graph grammar with injective matches, each rule $S : L \rightarrow R$ with $S \in \text{RULES}_0$ can be applied at most once with the same essential match $m_0 : L \in G_0$ and $m_0 \models NAC$.**

**Lemma 6.13. Termination of Nondeleting Layered Graph Grammars.** Each nondeleting layered graph grammar with injective matches terminates.


**Definition 6.15. Layer assignments.** $G_0$ is a start graph with start labels $T_0 \subseteq \text{LABELS}$ for each $l \in \text{LABELS}$ the creation and deletion layers can be defined as follows:

\[
\begin{align*}
cl(l) & = \text{if } l \in T_0 \text{ then } 0 \text{ else } \max\{rl(S) | S \text{ creates } l\} + 1 \\
dl(l) & = \text{if } l \text{ is deleted by some } S \text{ then } \min\{rl(S) | S \text{ deletes } l\} \text{ else } k_0
\end{align*}
\]
6.1.2 Termination Criteria for High-Level Replacement Units

When specifying transformations, it is hardly the case that a single, unstructured diagram rewriting system is used to define complex transformations. A typical problem is to steer the progress of the transformation towards some well-defined configuration of the diagram, i.e. state of the system. This may involve the definition of some sequence of rule applications, as well as the prevention of repeated application of the same rule to the same match, or of cyclic repetitions of the same sequence of applications.

In general, guaranteeing such properties of the rewriting process is equivalent to proving its termination, but following the classical approach of proving termination by constructing a monotone measure function on some multiset [42], and showing that the value of such a function decreases at each application. Further termination criteria use polynomial orderings or recursive path orderings [41].

In [23], a contribution towards solving the termination problem for rewriting systems with external control mechanisms is given. It extends the concept of transformation unit to high-level replacement systems. For high-level replacement units, several abstract properties based on termination criteria are stated and proved.

High-level replacement systems [57] [58] are a generalization of the graph transformation approach and fit into the general approach at the basis of the definition of transformation units [105]. The resulting notion is called high-level replacement unit. Its semantics is given by the set of all possible derivation sequences. Thereafter, high-level replacement units are instantiated by attributed graph transformation.

Let CAT be category with a distinguished morphism class M, such that CAT has pushouts and pullbacks along M-morphisms, i.e. if one of the given morphisms is in M, then also the opposite one is in M, and M-morphisms are closed under pushouts and pullbacks (the notation (CAT, M) also used to indicate CAT).

Definition 6.16. Control expressions. The class CA of control expressions over Names (representing a set of rule names) is recursively defined by

1. Names \⊆ CA,
2. \(C_1 ; C_2 \in CA\), if \(C_1, C_2 \in CA\),
3. \(C_1 | C_2 \in CA\), if \(C_1, C_2 \in CA\),
4. \(\text{asLongAsPossible } C \text{ end } \in CA\), if \(C \in CA\).

The intended meaning of the operator ; is the application of the expression \(C_1\) followed by the application of the expression \(C_2\). The operator choice (\(\_\)) allows the application of either the expression \(C_1\) or \(C_2\). Furthermore, the intended meaning of the operator \(\text{asLongAsPossible } C \text{ end }\) is the (sequential) application of the expression \(C\) as long as its application is possible.

Definition 6.17. High-level replacement unit. A high-level replacement unit \(RU = (RULES, nm, C)\) in a category \(CAT\), or just replacement unit, consists of a finite set RULES of rules, a bijective function \(nm : RULES \rightarrow Names\), and a control expression \(C \in CA\) over Names.
Definition 6.18. Termination criterion. A function $F : G \to \mathbb{N}$ from objects to natural numbers is a termination criterion for $(CAT, M)$ if for any two arbitrary morphisms $a : C \to A$ and $b : C \to B$ in $M$, the value $F(A +_C B)$ of the pushout object $A +_C B$ of $a$ and $b$ is given by $F(A +_C B) = F(A) + F(B) - F(C)$. Given a rule $S$ with morphisms in $M$, a termination criterion $F$ for $CAT$ is a termination criterion for $RULES$ if $F(L(S)) > F(R(S))$.

Proposition 6.19. If $F$ is a termination criterion for rule $S$, then it is also a termination criterion for all the derived rules $S_d$ of all $d : G \Rightarrow_p H$.

- If $F$ is a termination criterion for one derived rule $S_d$ of $d : G \Rightarrow_p H$, then it is a termination criterion for rule $S$.
- If $F$ is a termination criterion for $E_1$ and $E_2$, then it is also a termination criterion for $E = E_1; E_2$.
- If $F$ is a termination criterion for $E_1$ and $E_2$, then it is also a termination criterion for $E = E_1|E_2$.
- If $F$ is a termination criterion for $E_0$, it is also a termination criterion for $E = \text{asLongAsPossible } E_0 \text{ end}$.

Proposition 6.20. Termination of replacement units. Given a replacement unit $RU = (RULES, nm, C)$, all derivations terminate, if there is a termination criterion $F$ which holds for all $S \in RULES$.

Unfortunately, terminating rules do not always satisfy the measure function required by this approach, since attribute transformations and other constraint can also influence the termination properties of a rule.

6.1.3 Termination Criteria for DPO Transformations with Injective Matches

In [147] termination criteria are proposed for graph and model transformation systems with injective matches and finite input structure. A treatment is given for infinite sequences of rule applications that takes attribute conditions, negative application conditions, and type constraints into account.

Definition 6.21. An $E$-graph $EG = (V_G, V_D, E_G, E_{NA}, E_{EA}, (src_j, tar_j))_{j \in \{G, NA, EA\}}$ consists of graph and data nodes $V_G$ and $V_D$, and graph, node attribute and edge attribute edges $E_G$, $E_{NA}$, and $E_{EA}$, respectively. The domains and codomains of the source and target functions $src_j$ and $tar_j$ for the corresponding edges $E_j$ are depicted below.
Given a signature $DSIG = (S, OP)$ with attribute value sorts $S_D \subseteq S$, an attributed graph $AG = (EG, D)$ is an E-graph $EG$ together with a $DSIG$-algebra $D$ such that $V_D = \bigcup_{s \in S_D} D_s$.

Given an attributed graph $TG$ as type graph, a (typed attributed) graph $G = (AG, t)$ is an attributed graph $AG$ together with a typing morphism $t : AG \rightarrow TG$.

Typed attributed graphs and the corresponding morphisms form the category $\text{AGraphs}_{\text{ATG}}$.

The function to measure the size of a graph $G$ is defined as follows.

**Definition 6.22.** Given a graph $G = ((V_G, V_D, E_G, E_{NA}, E_{EA}, (src_j, tar_j))_{j \in \{G, NA, EA\}}, D)$, the size of $G$ is denoted by $|G|$ and calculated as follows: $|G| = |V_G| + |E_G| + |E_{NA}| + |E_{EA}|$. $G$ is finite if $|G| < \infty$.

Based on the concurrency theorem of the DPO approach the composition of two DPO rules as follows.

**Definition 6.23.** An $E$-concurrent production $p^*\!$ is an $E$-based composition if there is at least one input graph $G_0$ with an $E$-related transformation $G_0 \xrightarrow{p^*} H$.

This definition is required, because for the DPO approach, the definition of $E$-concurrent productions and the Concurrency Theorem have not been extended to handle negative application conditions and other constraints. Moreover, this definition guarantees, among others, that the constraints enforced by $p_1$ do not contradict to the constraints necessary for the application of $p_2$.

**Definition 6.24.** Given a possibly infinite sequence of graph productions $p_i, (i = 1, 2, \ldots)$ 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 $p_n^* = (p_1 * E_1 p_2) * E_2 \ldots * E_n p_n$.

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 $|L_n^*|$. It is possible that there are several cumulative LHS series of a given production sequence, since, in general, two rules can be composed in different ways, choosing different $E$-dependency relations.

**Lemma 6.25.** The sequence $|L_i^*|$ (Def. 6.24) is monotonic nondecreasing. If $E_i \cong R_i^*$, $L_i^*$ remains unchanged, thus, $|L_i^*| = |L_{i+1}^*|$. Otherwise, $L_i^* \neq L_{i+1}^*$ and $|L_i^*| < |L_{i+1}^*|$, but $L_{i+1}^*$ always contains an isomorphic subgraph of $L_i^*$.

**Proposition 6.26.** A $\text{GTS} = (P)$ (Def. 6.4) terminates if for all infinite cumulative LHS sequences $(L_i^*)$ of the graph productions created from the members of $P$, it holds that

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

*Note that finite input graphs and injective matches are assumed.*
The opposite direction of Proposition 6.26 does not hold in general, but for a finite number of input graphs. In this case, no infinite sequences of E-based compositions can be constructed.

**Proposition 6.27.** If a GTS = (P) (Def. 6.4) terminates and we have only a finite number of input graphs, then there are no infinite cumulative LHS sequences \( L^*_i \) of graph productions created from the members of \( P \).

From Proposition 6.26 the next statement follows:

**Lemma 6.28.** If \( L^*_i \not\equiv L^*_{i+1} \), \( \forall i \) for any cumulative LHS series (Def. 6.24), then the GTS terminates. If each graph appears only finitely many times in all cumulative LHS series, the GTS still terminates.

The contribution of this approach is to provide termination criteria for general productions allowing recursion within the scope of DPO and typed attributed graph transformation, assuming injective matches. This can be a theoretical basis to prove that certain control flows of rules are terminating, where the other - algorithmically underpinned - criteria cannot be applied. In general, however, it is hard to find all the possible sequences of graph productions, and prove that the corresponding series \( |L^*_i| \) exceeds all limits. This is expected, since the termination of a GTS is undecidable [166]. However, the stricter and the more deterministic the ordering of the rules is, the higher is the chance that we can deal with the sequences.

### 6.2 VMTS Visual Control Flow Language

One of the most important capabilities of a control flow language is the possibility to express a transformation as an ordered sequence of the transformation rules. Classical graph grammars apply any production that is feasible. This technique is appropriate for generating and matching languages but model-to-model transformations often need to follow an algorithm that requires a more strict control over the execution sequence of the rules, with the additional benefit of making the implementation more efficient.

The VMTS approach is a visual approach, and it also uses graphical notation for control flow: stereotyped activity diagram, which is a technique to describe procedural logic, business process, and work flow. In many ways, it plays a role similar to flowcharts, but the principal difference between it and the flowchart notation is that activity diagrams support parallel behavior [70].

An example Visual Control Flow Language (VCFL) model is depicted in Fig. 6.1 that is the VCFL transformation of the class model to relational database management system (RDBMS) model. This transformation is elaborated in Section 8.2.

In Fig. 6.2a, the metamodel of the VMTS control flow is depicted, which describes that the root element is the Transformation. A Transformation can contain any number of FlowEdgeTarget type object, this is denoted by stereotype << SystemContainment >>. The FlowEdgeTarget is an abstract type which could be Transaction, StartRule, Rule, HistoryRule, EndRule, FlowFinal, Decision, Merge, Fork or Join. FlowEdgeTargets can be connected to each other, using directed edges (FlowEdge). Types Transaction and Rule can contain another FlowEdgeTargets. Moreover, the type Rule can contain RuleNodes. This is
Fig. 6.1. VCFL model of the transformation Class2RDBMS

presented in the metamodel of the VMTS Rule Editor (Fig. 6.2b). RuleNode is also an abstract type that can be LHSNode or RHSNode. A type RuleNode can contain or can be connected to another RuleNodes.

The internal causality is a relation between LHS and RHS elements (Fig. 6.2b), it makes possible to connect an LHS element to an RHS element and to assign an operation to this connection. An internal causality describes what have to be done applying a transformation rule (element creation, element deletion, attribute modification). The create and the modify operations are accomplished by XSLT scripts. The XSLT scripts can access the attributes of the objects matched to LHS elements, and they produce a set of attributes for RHS element to which the causality point.

VCFL is a visual language for controlled graph rewriting and transformation that supports the following constructs: sequencing the transformation rules, branching with OCL constraints, hierarchical rules, parallel executions of the rules and iteration.

6.2.1 Sequencing Transformation Rules

Sequencing transformation rules results a transformation that contains the rules in an ordered sequence \( (S_0, S_1 \ldots S_{n-1} \in RULES) \). Assume the case that the input model of rule \( i \) \( (S_i) \) is the model \( G_i \) and the result of the \( S_i \) is the \( G_{i+1} \) (where \( 0 \leq i \leq n - 1 \)). In this case the input model of the rule \( i + 1 \) \( (S_{i+1}) \) is the model \( G_{i+1} \). This means that during the execution of the rule sequence, each rule works on the result of the previous rule. Obviously, except for the first
rule, which works on the input model. The result of the whole transformation is the result of the last rule ($S_n$).

The interface of the transformation rules allows the output of one rule to be the input of another rule, in a dataflow-like manner. This is used to sequence expression execution. In VCFL, this construction is referred to as external causality. An external causality creates a mapping between a node contained by RHS of the rule $i$ and a node contained by LHS of the rule $i + 1$. This feature accelerates the matching and reduces the complexity, because the rule $i$ can provide partial match to the rule $i + 1$.

### 6.2.2 Branching with OCL Constraints

Often, the transformation needs to be applied depending on a condition. Therefore, a branching construct is required. In VCFL, OCL constraints assigned to the decision elements can choose between the paths of optional numbers. The control flow path is chosen based on the properties of the actual input model and the success of the last transformation rule (built in variable $\text{SystemLastRuleSucceeded}$). If the last transformation rule fails, then VCFL could use the values of the system variables $\text{SystemLHSFailure}$ and $\text{SystemRHSFailure}$ for the decision. These variables represent whether a failure has occurred, because there was no proper match (LHS failure: structurally not suitable input model or there is at least one constraint not satisfied in LHS of the transformation rule), or the transformation result was not sufficient (RHS failure: there was at least one constraint not satisfied in RHS of the transformation rule).
In VCFL, each branch has an exact OCL guard condition. When a rule $S$ is connected to more than one follow-up rules, then at most one of the branch conditions is allowed to be true. This means that the conditions must not have any common part. This restriction ensures that the control flow execution of the VCFL is deterministic.

VCFL has been applied in projects such as generating user interface from resource model and user interface handler code from statechart diagram for mobile platform (Section 8.2). In addition, it has been used in several model-to-model transformation. These applications require control flow support, and all of them can be solved without non-determinism. However, VCFL provides an interface for nondeterministic control flow as well.

### 6.2.3 Hierarchical Rules

VCFL supports hierarchical specification of the transformation rules. High-level rules can be created by composing a sequence of primitive rules and can be viewed as separate transformation modules.

A high-level rule can contain several simple rules, hiding the details which could be unimportant on a specific abstraction level, and represents the contained rules as coherent units (Fig. 6.3).

Often, the OCL constraints assigned to a decision object do not cover all possible cases. It could result that none of the branch paths is selected in certain cases: the parent rule of the actual transformation handles the control flow: the engine breaks the execution of the transformation on the actual level and continues the transformation on the parent level.

### 6.2.4 Iteration, Tail Recursion and Parallel Executions of the Rules

The iteration is achieved with the help of the decision objects and the OCL constraints contained by them. A decision object evaluates the assigned constraints, and based on the results selects a flow edge which could be a follow-up or a backward edge as well (Fig. 6.1).

Recursion could be solved with the composition of the iteration and external causalities. A high-level rule can call itself, where external causalities represent the actual parameters of the recursive call.
Flattening the hierarchical state machine is an example when a recursive algorithm have to be applied that first calls flattening on its children before flattening itself [121].

The parallel execution of the independent transformation rules is supported by the Fork and Join elements.

In VCFL, if a transformation rule fails, and the next element in the control flow is a decision object then it could provide the next branch based on the OCL statements evaluated on the actual model and the value of the variable SystemLastRuleSucceed. If no decisions can be found, the control is transferred to the parent state, if there is no parent state, the transformation terminates with error.

### 6.3 Visual Control Flow Language Algorithms

VCFL provides algorithms to validate that the OCL constraints assigned to a decision object are disjoint and to check that the transformation rules contained by the transformation are not isolated or illegal.

The VCFL Isolated Transformation Rules algorithm checks whether the user-specified control flow contains isolated transformation rules. This means that, starting from the start rule, we cannot reach these rules. The algorithm checks the constraints contained by the decision objects whether all of the branches related to the actual decision object could be selected by the constraints. If a branch is found that can never be selected, the flow edge related to this branch is not taken into consideration by the algorithm. This means that not only the structure of the control flow model but the constraints contained by the decision objects are also taken into account to find isolated rules.

Firstly, checking the decision objects, the algorithm signs the invalid flow edges, secondly, identifies the isolated rules, using a modified breadth-first search. Transformation rules which are not found by the searching are the isolated rules. The pseudo code of the algorithm is as follows.

**Algorithm 6.2** Pseudo code of the VCFLIsolatedRules algorithm

1: VCFLISOLATEDRULES(VMTSTransformation $T$) : VMTSNodeCollection
2: for all decision in $T$ do
3:     for all constraint in decision do
4:         if NOTSuitableConstraint(constraint) then
5:             SignFlowEdgeByConstraint(constraint)
6:     SetVisited(startNode of $T$)
7:     ENQUEUE(queue, startNode of $T$)
8: while queue is not empty do
9:     node = DEQUEUE(queue)
10:    for all neighbor in GETNEIGHBORS(node) do
11:       if NOTVISITED(neighbor) then
12:          SetVisited(neighbor)
13:          ENQUEUE(queue, neighbor)
14: return GETNOTVISITEDNODES($T$)
<table>
<thead>
<tr>
<th>Type</th>
<th>Notation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision</td>
<td><img src="image" alt="Decision" /></td>
<td>A decision is a control node that chooses between outgoing flows.</td>
</tr>
<tr>
<td>EndRule</td>
<td><img src="image" alt="EndRule" /></td>
<td>An end rule is a final node that stops all flows in a transformation.</td>
</tr>
<tr>
<td>FlowEdge</td>
<td><img src="image" alt="FlowEdge" /></td>
<td>A flow edge is a directed connection between two rules.</td>
</tr>
<tr>
<td>FlowFinal</td>
<td><img src="image" alt="FlowFinal" /></td>
<td>A flow final is a final node that terminates a transformation.</td>
</tr>
<tr>
<td>Fork</td>
<td><img src="image" alt="Fork" /></td>
<td>A fork is a control node that splits a flow into multiple concurrent flows.</td>
</tr>
<tr>
<td>HistoryRule</td>
<td><img src="image" alt="HistoryRule" /></td>
<td>A history rule represents a composite transformation rule that stores the most recent active configuration of the transformation rule.</td>
</tr>
<tr>
<td>Join</td>
<td><img src="image" alt="Join" /></td>
<td>A join node is a control node that synchronizes multiple flows.</td>
</tr>
<tr>
<td>Merge</td>
<td><img src="image" alt="Merge" /></td>
<td>A merge is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.</td>
</tr>
</tbody>
</table>
| Rule        | ![Rule](image) | A rule represents a simple or composite transformation rule in the transform.
| StartRule   | ![StartRule](image) | A start rule is a control node at which flow starts when the transformation is invoked. |

The VCFL Illegal Transformation Rules algorithm detects rules in control flow models from which no EndRule or FlowFinal can be reached. The algorithm is similar to the VCFL Isolated Transformation Rules algorithm with the following difference. The modified breadth-first search is started from EndRules and FlowFinals, and uses the edges in the reverse direction as they are in the control flow model. Transformation rules which are not found by the algorithm are the rules from which the end rules are unreachable.
The VCFL Disjoint OCL Constraint algorithm validates whether the OCL constraints assigned to a decision object are disjoint. This algorithm ensures that at the same time maximum one of the branch conditions of a decision is allowed to be true. Using this algorithm it is guaranteed that the control flow execution of the VCFL is deterministic. The algorithm utilizes that the OCL statements are boolean expressions. It does an and operation for each couple of the OCL statements, and if the result is false in each cases, then only one of the OCL statements could be true at the same time.

Algorithm 6.3 Pseudo code of the VCFLDisjointConstraint algorithm

1: VCFLDisjointConstraint (VCFLDecision D) : VMTSConstraintPairList
2: for all constraintA in D do
3:   for all constraintB in D do
4:     if constraintA != constraintB and DoANDOnConstraints(constraintA, constraintB) then
5:       AddToList(constraintPairList, constraintA, constraintB)
6: return constraintPairList

The most complex and maybe the most important VCFL algorithm is the VCFL Termination algorithm which is discussed in Section 6.5.

6.4 Composing Metamodel-Based Model Transformation Rules

This section provides considerations and algorithms to compose metamodel-based model transformation rules. Rule composition facilitates to predict the behavior of a transformation, for example, to support the termination analysis of transformations.

On the instance layer, the transformation rule composition is constructed in [54], based on Concurrency Theorem [51] of the double pushout approach. The goal of this section is to develop an equivalent transformation rule composition method on the layer of the metamodel-based transformation rules.

In order to examine the termination properties of VCFL exhaustive transformation rules and VCFL loops, a transformation rule composition mechanism is applied. This is a way to examine all the possible transformation execution without the actual input models. The composed rule \( S_C \) can equivalently replace the original rules \( S_j, S_{j+1} \ldots S_k \in RULES \), because it produces the same result and imposes the same input conditions as the sequence of individual rules [54]. In the termination analysis, the composed rule can be used instead of the original transformation rules. It facilitates to replace the rules contained by a VCFL loop with their composed transformation rule. The result of the replacement is similar to an exhaustive transformation rule, with the difference that a composed rule may have a decision object.

Moreover, to examine the termination properties of the exhaustive transformation rules the composition algorithm can also be used. Recursively composing an exhaustively applied transformation rule \( S_E \) with itself \( n \) times results a transformation rule \( S^n_E \) that can be used once for an input model \( G_0 \) instead of applying the original rule \( S_E \) \( n \) times.
The composition algorithm takes not only the structure of the rules into consideration but also the external and internal causalities, and the metatypes of the PRNs and edges.

In general, the $E$-based composition of the transformation rules algorithmically explodes, because there are infinite number of possible compositions.

The external causalities defined between the rules simplify the complexity of the rule composition. They exactly define the mapping between RHS elements of the rule $i$ and LHS elements of the rule $i + 1$.

Internal causalities connect LHS and RHS elements within a transformation rule. Therefore, taking both the internal- and external causalities into account, the node mapping can be followed between the transformation rules and also within them. Thus, the PRNs can be identified unambiguously, and followed through a loop or a whole transformation.

The metatypes of the PRNs and edges, compared to the general case, also simplify the computation complexity of the algorithm. The metatypes reduces the search space. The worst case is, when all the PRNs have the same metatype. In case of metamodel-based transformation rule composition external causalities provide an initial mapping, which must be extended, based on the metatype-based mapping. All possible cases and compositions must be examined and considered as a possible mapping. If there is no external causality defined, the algorithm starts from a node with metatype which has the fewest occurrences in $S_{i}^{RHS}$ and $S_{i+1}^{LHS}$.

In Fig. 6.4, the $E$-based composition of the transformation rules is completed with the metamodels of LHS and RHS of the transformation rules ($LM_1$, $RM_1$, $LM_2$, and $RM_2$), the metamodels of the processed and resulted models ($GM_1$, $HM$, and $GM_2$), and instantiation relations ($lm_1$, $rm_1$, $lm_2$, $rm_2$, $gm_1$, $hm$, and $gm_2$). Furthermore, $L$ and $R$ denote LHS and RHS of the composed rule, $LM$ and $RM$ are their metamodels, and $lm$ and $rm$ are instantiations. The interface graph of the composed rule ($K$) is not calculated now, because it is not relevant related to the metmodel-based model transformation rule composition.
Chapter 6. Control Flow Related Properties of Model Transformations

The E-based composition of metamodel-based model transformation rules (EBMR) depends on the relation between $RM_1$ and $LM_2$.

**Proposition 6.29.** $p_1 = (L_1 \xrightarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xrightarrow{l_2} K_2 \xrightarrow{r_2} R_2)$ are two sequentially executed transformation rules, and $LM_1$, $RM_1$, $LM_2$, and $RM_2$ are the metamodels of $p_1^{LHS}$, $p_1^{RHS}$, $p_2^{LHS}$, and $p_2^{RHS}$ respectively. Furthermore, $L$ is LHS and $R$ is RHS of the E-based composition of $p_1$ and $p_2$ ($p_1 * E p_2$) with metamodels $LM$ and $RM$, as presented in Fig. 6.4. If there is no common part of $RM_1$ and $LM_2$, - there is no metatype that is included by both metamodels ($\forall T_1 \in RM_1 \land T_1 \notin LM_2 \land \forall T_2 \in LM_2 \land T_2 \notin RM_1$) -, then there is no E-based composition of the metamodel-based model transformation rules (EBMR) ($E$ graph is empty).

**Proof.** The metamodels $RM_1$ and $LM_2$ do not contain any node with the same metatype. Therefore, there are no nodes in the instances ($R_1$ and $L_2$) with the same metatype that can be mapped to each other. Thus it is not possible to compose the rules, consequently there is no EBMR. □

If $RM_1$ and $LM_2$ have common part, then rules can be composed, and in general, there are not only one but several ways to compose them.

### 6.4.1 Self-Composing Metamodel-Based Model Transformation Rules

This section examines how to compose transformation rules with themselves any times. This type of composition is required, for example, to examine the termination of the exhaustively applied transformation rules. The section provides several examples and an algorithm composing transformation rules. In order to state and prove the general case formally, the composition properties of several typical transformation rule structures need to be examined. The examined structures are the *tree*, *inserting* and *deletion structure* rules. The following examples and considerations discuss the properties of these structures and the possibilities of their composition with themselves. This type of examination of the transformation rule structures is chosen, because any transformation rule is built from these structures. Therefore, if the correctness of these structures is shown and proven one-by-one, then it makes easier to generalize the considerations for the composition of the general transformation rules.

The following example structures are exhaustively executed transformation rules. The structures are so simple that there is only one possible composition between the nodes of $S_i^{RHS}$ and $S_{i+1}^{LHS}$, but external causalities should be defined in order to achieve that the composed transformation rule performs the same modification on the instance layer as the original rules.

**Definition 6.30.** The nodes generated and attached to the input model $G_0$ by a *tree structure transformation rule* $S_T$ are connected only with one edge to the input model.

A *tree structure* transformation rule does not generate circles into the input model $G_0$. An example transformation rule with *tree structure* is depicted in Fig. 6.5a. The first application of the rule on an input model is presented in Fig. 6.5b/1. It matches an $A$ type node with three $B$ type nodes and generates three $C$ type nodes into the model. A possible application of the transformation rule (Fig. 6.5b/2) is executed on the result of the first execution. In Fig. 6.5,
Fig. 6.5. Self-composition of tree structure transformation rules

the dashed lines denote the matched part of the input model and the created part of the output model. If there are no external causalities defined, then in the second application of the rule the matching algorithm can find the same A and B type nodes, but it is also possible that only a part of them with new nodes or totally new nodes are found. This means that the execution of the rules defined without external causalities can become ambiguous on the instance layer. The algorithm that VMTS uses to compose metamodel-based model transformation rules with themselves is as follows.

Based on the external causalities, during the transformation rule composition we have to calculate the multiplicities in two different ways. The exact rules for calculating the multiplicities during transformation rule composition are summarized in Table 6.2.

- If there are no external causalities defined the transformation rule presented in Fig. 6.5a is composed with itself once and the result is depicted in Fig. 6.5c. In the current case, the composed rule differs from the original one only in the modified multiplicities. The multiplicities of the matched and the created nodes vary between well-defined limits. Multiplicities must allow the cardinality values between \(m\) and \(n \times m\), where \(m\) denotes the cardinality of the original multiplicity, and \(n\) the number of the application of the rule.
Chapter 6. Control Flow Related Properties of Model Transformations

Algorithm 6.4 Pseudo code of the VTSRECURSIVECOMPOSING algorithm

```python
1: VTSRECURSIVECOMPOSING(TransformationRule originalRule, TransformationRule[] inComposedRules, int numOfComposition): TransformationRule[
2:   if numOfComposition == 0 then
3:     return inComposedRules
4:   for all TransformationRule S in inComposedRules do
5:     initialMapping = CREATEMAPPINGBYEXTERNALCAUSALITIES(S, originalRule)
6:     initialMapping = EXTENDMAPPINGBYMETATYPES(initialMapping, S, originalRule)
7:     newCompositions = CREATELHSANDRHS(initialMapping, S, originalRule)
8:     CALCULATEMULTICITIES(newCompositions)
9:     AddToList(newCompositionList, newCompositions)
10: return VTSRECURSIVECOMPOSING(originalRule, newCompositionList, numOfComposition - 1)
```

Table 6.2. Rules for calculating multiplicities during transformation rule composition

<table>
<thead>
<tr>
<th>Multiplicity in rule $S_1$</th>
<th>Multiplicity in rule $S_2$</th>
<th>Multiplicity in composed rule without external causalities</th>
<th>Multiplicity in composed rule with external causalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$r$</td>
<td>$\max(p, r).p + r$</td>
<td>$\min(p, r).\max(p, r)$</td>
</tr>
<tr>
<td>$p.s$</td>
<td>$r$</td>
<td>$\max(p, r).s + r$</td>
<td>$\min(p, r).\max(s, r)$</td>
</tr>
<tr>
<td>$p_1.s_1$</td>
<td>$p_2.s_2$</td>
<td>$\max(p_1, p_2).s_1 + s_2$</td>
<td>$\min(p_1, p_2).\max(s_1, s_2)$</td>
</tr>
</tbody>
</table>

- If there are external causalities defined between nodes $A \in S_{tRH}^R$ and $A \in S_{tL}^L$, furthermore, between $B \in S_{tRH}^R$ and $B \in S_{tL}^L$, then the multiplicities during the composition should be calculated based on the fourth column of the Table 6.2. This external causalities specify that the exhaustively executed transformation rule should match the same nodes with type $A$ and $B$ on the instance layer. The composed transformation rule for this case is presented in Fig. 6.5d.

In VMTS, if the interval or * multiplicity is allowed, then the match found for LHS in the finite input model ($G_0$) 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 do not contradict to the other part of the match. It means that no more links can be added to the match, which is also a match for a particular LHS.

In Fig. 6.6 two transformation rules $S_1$ and $S_2$ with an external causality between nodes $A \in S_{tRH}^R$ and $A \in S_{tL}^L$ (Fig. 6.6a and Fig. 6.6b) and their composed rule (Fig. 6.6c) are depicted. If the first rule ($S_1$) matches $h_1|h_1 < s_1$ B type node, then $S_2$ cannot match a $B$ type node that is not matched by rule $S_1$. Based on the previous paragraph if there are more $B$ type nodes than $h_1$ connected to the actually matched $A$ type node, then $S_1$ should match them. Rule $S_2$ can extend the number of the matched $B$ type nodes only if rule $S_1$ matches the maximally allowed amount of $B$ type nodes ($s_1$) and of course $s_2 > s_1$. If rule $S_1$ matches $h_1|h_1 < s_1$ $B$ type nodes, then $S_2$ cannot extend it. Therefore, the composed rule (Fig. 6.6c) with the multiplicities calculated based on Table 6.2 has the same effect on the instance layer as rules $S_1$ and $S_2$ executed one after another.
Chapter 6. Control Flow Related Properties of Model Transformations

The multiplicities appearing in RHS vary between well-defined limits in order to cover all possible outputs that can belong to the valid input models. It does not mean any ambiguity because the operations are defined exactly by the internal causalities of the transformation rules.

Remark 6.31. Completion for Table 6.2: if RHS of the composed rule contains circle or create type internal causality for the actual PRN, then the multiplicity can exceed the calculated amount.

Proposition 6.32. Let $S_T$ be a tree structure transformation rule with external causalities linking PRNs appearing both in RHS and LHS, and let $S_T^n$ be the transformation rule resulted by composing the rule $S_T$ $n$ times, using VTSRecursiveComposing algorithm. Applying the tree structure transformation rule $S_T^n$ times on an arbitrary finite input model $G_0$ is equivalent to a single execution of rule $S_T^n$. 

Proof. Appendix A.

Definition 6.33. The nodes generated and attached to the input model $G_0$ by an inserting structure transformation rule $S_I$ are connected at least by two generated edges to the model $G_0$.

If the input nodes are connected, the inserting structure transformation rule $S_I$ generates circle into the model $G_0$. An example for the inserting structure transformation rule is depicted in Fig. 6.7a. For the case when there are no external causalities defined, a first and a second possible application on an input model of the rule is presented in Fig 6.7b. The transformation rule presented in Fig. 6.7a is composed with itself once, and the result is depicted in Fig. 6.7c.

Two cases must be differentiated. (i) The inserting structure transformation rule deletes the edge between the matched nodes that are connected through the newly created node (Fig. 6.7a). In this case the match is destroyed and cannot be found again. Consequently, it looses the sense of the exact external causality definition, because the match cannot be found again in the next rule. (ii) In the second case the rule does not delete the original edge that connects the nodes which are linked also through the newly created node and edges (Fig. 6.8a).
Proposition 6.34. Let $S_I$ be an inserting structure transformation rule with external causalities linking PRNs appearing both in RHS and LHS, and let $S'_I$ be the transformation rule resulted by composing the rule $S_I$ $n$ times, using VTSRecursiveComposing algorithm. Applying the inserting structure transformation rule $S_I$ $n$ times on an arbitrary finite input model $G_0$ is equivalent to a single execution of the rule $S'_I$.

Proof. Appendix A.

Definition 6.35. A deleting structure transformation rule $S_D$ deletes at least one node from the input model $G_0$.

An example for a metamodel-based deleting structure transformation rule is depicted in Fig. 6.8b/1, it is composed with itself, and for the case when there are no external causalities defined, the resulted rule is presented in Fig. 6.8b/2. Fig. 6.8d depicts the composed transformation rule if there are external causalities defined between nodes $A \in S'^{RHS}_D$ and $A \in S'^{LHS}_D$.

Proposition 6.36. Let $S_D$ be a deleting structure transformation rule with external causalities linking PRNs appearing both in RHS and LHS, and let $S''_D$ be the transformation rule resulted by composing the rule $S_D$ $n$ times, using VTSRecursiveComposing algorithm. Applying the deleting structure transformation rule $S_D$ $n$ times on an arbitrary finite input model $G_0$ is equivalent to a single execution of the rule $S''_D$. 
Proposition 6.37. A tree, inserting or deleting structure transformation rule $S$ does not contain external causalities. Let $S^n$ be the transformation rule resulted by self-composing rule $S$ $n$ times. Applying the rule $S^n$ once can produce different result as $n$ times execution of rule $S$. This means their execution is ambiguous on the instance layer.

Proof. Appendix A.
Chapter 6. Control Flow Related Properties of Model Transformations

Algorithm 6.5 Pseudo code of the VTSComposing algorithm

1: VTSComposing(TransformationRule[], TSs, bool exhaustive): TransformationRule[]
2: compositionList = GetFirstRule(TSs)
3: if exhaustive then
4: return VTSRecursiveComposing(GetFirstRule(TSs))
5: else
6: for all TransformationRule ruleNext in TSs (except first rule) do
7: for all TransformationRule S in compositionList do
8: initialMapping = CreateMappingByExternalCausalities(S, ruleNext)
9: initialMapping = ExtendMappingByMetatypes(initialMapping, S, ruleNext)
10: newCompositions = CreateLHSAndRHS(initialMapping, S, ruleNext)
11: CalculateMultiplicities(newCompositions)
12: AddToList(newCompositionList, newCompositions)
13: compositionList = newCompositionList
14: return compositionList

Proposition 6.38. \( p_1 = (L_1 \xrightarrow{l_1} K_1 \xrightarrow{r_1} R_1) \) and \( p_2 = (L_2 \xrightarrow{l_2} K_2 \xrightarrow{r_2} R_2) \) are two sequentially executed transformation rules, and \( L_{M_1}, R_{M_1}, L_{M_2}, \) and \( R_{M_2} \) are the metamodels of \( p_1^{LHS}, p_1^{RHS}, p_2^{LHS}, \) and \( p_2^{RHS} \) respectively. Furthermore, L is LHS and R is RHS of the E-based composition of \( p_1 \) and \( p_2 \) (\( p_1 \ast E p_2 \)) with metamodels LM and RM, as presented in Fig. 6.4. Let \( S_1 \) and \( S_2 \) be two metamodel-based model transformation rule that \( S_{LHS} = L_{M_1}, S_{RHS} = R_{M_1}, S_{LHS} = L_{M_2}, \) and \( S_{RHS} = R_{M_2} \). If \( S_C \) is the composition of the rules \( S_1 \) and \( S_2 \), using VTSComposing algorithm, then \( S_C^{LHS} \) is the metamodel of L (\( S_C^{LHS} = L_{M} \)) and \( S_C^{RHS} \) is the metamodel of R (\( S_C^{RHS} = R_{M} \)).

Proof. Appendix A.

Definition 6.39. Two metamodel-based model transformation rules have a unique composition if they can be composed exactly one way.

Proposition 6.40. The composition of two metamodel-based model transformation rules \( S_1 \) and \( S_2 \) is unique if there is exactly one \( E \) graph on the instance layer.

Proof. If there is exactly one \( E \) graph on the instance layer, then there is exactly one way to map \( S_1^{RHS} \) to \( S_2^{LHS} \) on the metamodel layer that corresponds to the Def. 6.39.

External causalities defined between transformation rule elements help to decrease the number of the possible compositions or to achieve the unique composition.

The following proposition states that there is no difference between applying a sequence of transformation rules or their unique composed rule (generated with VTSComposing algorithm) for an arbitrary input model. The two transformations result the same output model.

Proposition 6.41. If the transformation rules \( S_j, S_{j+1}, \ldots, S_k \in RULES \) are applicable successfully for a finite input model \( G_0 \), \( S_C \) is a unique composed transformation rule that can composed from transformation rules \( S_j, S_{j+1}, \ldots, S_k \) using VTSComposing algorithm, then it has the same effect on the input model \( G_0 \) as the transformation rules \( S_j, S_{j+1}, \ldots, S_k \).
Proof. It follows from Propositions 6.32, 6.38 and 6.40.

Utilizing the presented metamodel-based model transformation rule composing approach the next section examines the termination properties of VCFL transformations.

6.5 VCFL Termination Properties

The termination properties of model transformations are really important. It must be investigated under which conditions an arbitrary VCFL transformation can satisfy termination criteria. Recall from Section 4.2.2.1 that the difference between a transformation and a finite sequence of rules is that a finite sequence of rules always terminates, but a transformation, can contain infinite number of steps. The aim is that VCFL transformations terminate, therefore an algorithm has been developed to support the early detection of the infinite loop and the validation of the control flow that from each rule can reach an end rule.

In VCFL, a transformation rule has two specific attributes: Exhaustive and MultipleMatch. Recall that applying a model transformation rule means finding a match of LHS in the input model and replacing this submodel with RHS. An exhaustive transformation rule is executed continuously as long as LHS of the rule could be matched to the input model. The MultipleMatch attribute of a rule allows the matching process to find all occurrences of LHS in the input model, and replacing is executed on all the found places.

Definition 6.42. A VCFL transformation is a stereotyped UML activity diagram. A VCFL transformation $T$ defines a strict order of the contained transformation rules $S_0, S_1, ..., S_{n-1} \in RULES \in T$, where $S_0$ is the start rule of the $T$. Transformation $T$ contains OCL constraints, assigned to decision objects to choose between different control flow branches and external causalities between transformation rules to support parameter passing.

Definition 6.43. A VCFL transformation $T$ for a finite input model $G_0$ terminates, if there is no infinite derivation sequence from $G_0$ via transformation rules $RULES \in T$, where starting from $S_0$ rules $RULES$ are applied as it is defined by the transformation $T$.

For non-exhaustive and also for exhaustive transformation rules, the MultipleMatch attribute of the rules does not modify the termination property of the VCFL control flows for any finite input model $G_0$.

During the validation of the termination properties of a transformation it is necessary to be taken into account whether the VCFL transformation contains loops with decision objects or exhaustive transformation rules.

6.5.1 VCFL Control Flows with Non-Exhaustive Transformation Rules

Proposition 6.44. A VCFL transformation $T$ which contains only non-exhaustive transformation rules $S_0, S_1, ..., S_{n-1} \in RULES$ and does not contain loops for any finite input model $G_0$ always terminates.
Proof. The transformation $T$ contains finite number of transformation rules ($n = \#RULES \land n < \infty$). $\forall i\{0 \leq i \leq n - 1 \} S_i \in RULES$ is executed at most once because it is a non-exhaustive rule.

If the multiple match attribute of a rule $S_i \in RULES$ is true, all occurrence of the $S_i^{LHS}$ is searched and the replacement is executed for all found matches. The number of the found matches ($m_i$) is also finite because of the finite input model $G_0$. $n < \infty \land m_i < \infty | 0 \leq i \leq n - 1$, therefore $k = \sum_{i=0}^{n-1} m_i < \infty$. The number of the rules executed by transformation $T$ is finite and $T$ terminates.

6.5.2 VCFL Control Flows with Exhaustive Transformation Rules

Definition 6.45. $G_m \subseteq G_n$ if and only if $G_n$ has a structurally isomorphic subgraph $G_I$ to $G_m$, and in the $G_I$ and in the $G_m$ the corresponding nodes and edges have the same metatype, attributes, attribute values and OCL constraints.

An exhaustively applied rule using external causalities gives itself the input model and the parameters. For an exhaustive rule, the termination algorithm must take into consideration the attribute modifications, as well as the generated and deleted elements. An exhaustive transformation rule must contain either an attribute modification or an element deletion to prevent that the same match is found again and again by the matching process. A solution can be also if there is a create type causality and an OCL constraint which holds before the creation and becomes false afterwards, therefore, it prevents to find the same match again at the same place. For example, an OCL constraint can validate the existence of a neighbor node. In Fig. 6.9 the presented transformation rule connects a married and unemployed person to a company. The unemployed property is checked by the constraint $\text{const\_employer}$. After the execution of the rule, the matching process cannot match the same pattern again in the next iteration, because of the not satisfied constraint. Thus it forbids the repeated application of the same rule at the same place again.

Definition 6.46. A terminating creation rule $S_{TC}$ contains an OCL constraint $C$ in $S_{TC}^{LHS}$, which must stand for the input models matched to the $S_{TC}^{LHS}$, and, as a result of the rule execution, the condition required by the constraint $C$ becomes false. Furthermore, if the rule
$S_{TC}$ can be composed with itself, and it holds for each $E$-based composition ($S_C = S *_{E_1} S *_{E_2} \ldots *_{E_{n-1}} S$), then for all possible composition $S_i^{RHS}$ is not included by the $S_i^{LHS}$.

**Proposition 6.47.** An exhaustively applied TC rule $S_{TC}$ terminates for arbitrary finite input model $G_0$.

**Proof.** Executing rule $S_{TC}$ results that the condition required by the constraint $C$ becomes false in the actual match, therefore the same match cannot be found again. The TC rule $S_{TC}$ does not generate input for itself, the input model $G_0$ is finite, therefore, the exhaustively applied TC rule $S_{TC}$ terminates.

The following propositions contain statements about termination properties of the transformations with exhaustive transformation rules.

**Proposition 6.48.** Let the transformation rule $S_i \in RULES$ be an exhaustive rule. If $S_i^{LHS} \subseteq S_i^{RHS}$ and the rule $S_i$ has a match $M$ on any input model $G_i$ the rule $S_i$ never terminates for the input model $G_i$.

**Proof.** The rule $S_i$ has a match $M$ on the input model $G_i$ it generates its output ($G_i'$) with the $S_i^{RHS}$, $S_i^{LHS} \subseteq S_i^{RHS}$, therefore the $S_i^{LHS}$ has match in $G_i'$. The rule $S_i$ is an exhaustive rule and it always has match on the result model of the previous iteration, therefore the $S_i$ never terminates for the input model $G_i$.

**Proposition 6.49.** Let the transformation rule $S_i \in RULES$ be an exhaustive rule which does not contain internal causalities of type deletion and modification, and $S_i$ is not a TC rule. Assume that $T$ is a transformation and $S_i \in T$, the input model of the transformation $T$ is the model $G_0$, and the input model of the rule $S_i$ is the model $G_i$. If $S_i^{LHS}$ has a match $M$ on model $G_i$, the transformation $T$ never terminates for the input model $G_0$.

**Proof.** Rule $S_i$ is an exhaustive transformation rule, it is executed as long as $S_i^{LHS}$ has a match on the model $G_i$. $S_i$ has a match $M$, which is not modified by the rule - there is no deletion, attribute modification, and $S_i$ is not a TC rule -, therefore the matching process finds the match $M$ in each iteration. Rule $S_i$ never terminates for the input model $G_i$, and $T$ never terminates for the input model $G_0$.

### 6.5.3 Termination Properties of VCFL Loops

A loop $L$ contains $n$ transformation rules $S_h, S_{h+1} \ldots S_{h+n} \in RULES$ (where $n > 0$) and a decision object. A decision object evaluates the assigned constraints on the actual input model, and, based on the results, it selects a flow edge which could be a forward or a backward edge as well.

The main difference between a loop with only non-exhaustive rules and an exhaustive rule is the exit condition. A transformation leaves an exhaustive rule if there is no more match, while in the case of a loop the decision object determines about the exit. If a loop consists of non-exhaustive rules, the rule composition algorithm composes them, and the decision about the termination is made based on the composed rule and the OCL constraints of the decision object.
An exhaustive rule is itself a specific loop. Therefore, if a loop contains exhaustive rules then it is a loop of loops. The algorithm examines separately the exhaustive rules and if each of them terminates then analyses the whole loop. The termination of a loop \( L \) depends on the constraints (exit conditions) contained by its decision object.

**Proposition 6.50.** Assume that the transformation \( T \) contains a loop \( L \), let \( S_C \) be the composition of the non-exhaustive transformation rules \( S_h, S_{h+1} \ldots S_{h+n} \in L \). Also assume that the only exit condition of the loops decision object is not SystemLastRuleSucceed. The input model of the transformation \( T \) is the model \( G_0 \), and the input model of the rule \( S_C \) is the model \( G_C \). If \( S_C^{LHS} \subseteq S_C^{RHS} \) and the rule \( S_C \) has a match \( M \) on input model \( G_C \) the transformation \( T \) never terminates for the input model \( G_0 \).

**Proof.** The transformation rule \( S_C \) has a match \( M \) on the input model \( G_C \), it generates its output model \( G'_C | S_C^{RHS} \subseteq G'_C \). \( S_C^{LHS} \subseteq S_C^{RHS} \), therefore the \( S_C^{LHS} \) has match on model \( G'_C \). The rule \( S_C \) represents a loop and it always has match on the result model of the previous iteration, therefore the \( S_C \) never terminates for the input model \( G_C \) and the transformation \( T \) never terminates for the input model \( G_0 \). \( \square \)

### 6.5.4 VCFL Termination Algorithm

For any VCFL transformation \( T \), the termination algorithm validates the following.

- If transformation \( T \) does not contain loop or exhaustive transformation rule, then \( T \) terminates.
- If \( S_i \in T \) is an exhaustive transformation rule and \( S_i^{LHS} \subseteq S_i^{RHS} \), the transformation \( T \) does not terminate.
- If \( S_i \in T \) is an exhaustive transformation rule, \( S_i \) does not contain delete and modify type internal causalities, and \( S_i \) is not a TC rule, then the transformation \( T \) does not terminate.
- If \( L \in T \) is a loop and \( S_C \) is the composition of the transformation rules \( S_h, S_{h+1} \ldots S_{h+n} \in L \), \( S_C^{LHS} \subseteq S_C^{RHS} \), the only exit condition of \( L \) is not SystemLastRuleSucceed, then the transformation \( T \) does not terminate.

In order not to get into infinite loop the algorithm checks only the first \( N \) composition of the rules (where \( N \) is configurable).

If a transformation rule \( S \) contains create type internal causality, the algorithm checks whether the input model with the newly added elements contains new possible match places. The algorithm takes the structure of the pattern, metatypes of the nodes and edges, their attributes and attribute values and also the propagated OCL constraints into consideration.

During the composition of rules \( S_i \) and \( S_{i+1} \), the \( S_i^{RHS} \) and the \( S_{i+1}^{LHS} \) can have more than one matching variation. The algorithm checks all the possible variations taking into account external causalities, metatypes and constraints.

In the case of loops, the exit conditions (structure, attribute values and SystemLastRuleSucceed) are also checked by the algorithm.
Algorithm 6.6 Pseudo code of the VCLTermination algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VCFLTermination(VMTSTransformation T, N): retValue</td>
</tr>
<tr>
<td>2</td>
<td>if T does not contain loop or exhaustive rule then</td>
</tr>
<tr>
<td>3</td>
<td>return retValue.true</td>
</tr>
<tr>
<td>4</td>
<td>for all Transformation Rule S in T do</td>
</tr>
<tr>
<td>5</td>
<td>if S is exhaustive and S^{RHS} \in S^{LHS} then</td>
</tr>
<tr>
<td>6</td>
<td>return retValue.false</td>
</tr>
<tr>
<td>7</td>
<td>if S is exhaustive and S does not contain modify or deletion and S is not an TC rule then</td>
</tr>
<tr>
<td>8</td>
<td>return retValue.false</td>
</tr>
<tr>
<td>9</td>
<td>for all Loop L in T do</td>
</tr>
<tr>
<td>10</td>
<td>if counter ++ &gt; N then</td>
</tr>
<tr>
<td>11</td>
<td>return retValue.undecided</td>
</tr>
<tr>
<td>12</td>
<td>if exit condition == not SystemLastRuleSucceed then</td>
</tr>
<tr>
<td>13</td>
<td>composedRules = COMPOSETRANSFORMATIONRULES(transformation rules of the L)</td>
</tr>
<tr>
<td>14</td>
<td>for all rule in composedRules do</td>
</tr>
<tr>
<td>15</td>
<td>if rule^{RHS} \in rule^{LHS} then</td>
</tr>
<tr>
<td>16</td>
<td>return retValue.false</td>
</tr>
<tr>
<td>17</td>
<td>return retValue.undecided</td>
</tr>
</tbody>
</table>

VTA is an offline algorithm: the termination in many cases depends not only on the VCFL transformation model but also on the actual input model. A simple constraint could be itself a significant difference between two rules or models. The problem is not trivial: there are cases when the algorithm can make a decision based on the VCFL transformation for certain, and there are other cases when not.

6.6 Example Transformation and Its Termination Properties

This section introduces the practical relevance of the presented approach on the case study Class2RDBMS. Here, only a part of the whole transformation is presented that is necessary to illustrate the rule composition and termination criteria.

During the transformation process, helper nodes can be used in VMTS, which can connect two nodes of any type. This means that the metatypes of the connected nodes can even belong to different metamodels. These helper nodes can also contain any attributes. A << SystemAtom >> node is created with two edges pointing to the two connected objects. However, for the sake of simplicity, this construct is referred to as a helper node.

The control flow model of the whole transformation is depicted in Fig. 6.1. The control flow can be divided into three parts according to the goal of the units. (i) The large loop on the top (CreateTable, CreateParentClassHelper, AddParentAssociation, ShiftParentClassHelper, DeleteParentClassHelper) is responsible for the table creation and inheritance-related issues. (ii) The rule ProcessAssociation processes the associations. (iii) Finally, the last rules remove the helper nodes and temporary associations.

One of the major challenges is to process the inheritance hierarchy properly.
Fig. 6.10. Transformation rule AddParentAssociation

Rule AddParentAssociation (Fig. 6.10) creates a temporary association (tempAssociation) that links the subclass (Class) to the neighbors (NeighborClass) of the parent class (ParentClass). These associations facilitate that the rule ProcessAssociations processes not only the direct associations of a class, but the association of its parents as well.

The ParentClassHelperNode connects a subclass with its parent class, but the parent class can also have a parent. The transformation must traverse the whole inheritance hierarchy. The rule ShiftParentClassHelper (Fig. 6.11) removes the original ParentClassHelperNode and adds a new one, which links the subclass to the parent of the parent class. If the execution of the rule is successful (the parent class also has a parent), then the control is passed to the rule AddParentAssociation, otherwise to the rule DeleteParentClassHelper. The decision object makes the decision according to the system variable SystemLastRuleSucceed, which contains the result of the previous rule (ShiftParentClassHelper) execution.
Chapter 6. Control Flow Related Properties of Model Transformations

Fig. 6.12. External causalities between rules AddParentAssociation and ShiftParentClassHelper

Fig. 6.13. The E-based composition of the composed metamodel-based model transformation rules: ShiftParentClassHelper \( \ast_E \) AddParentAssociation

The rules AddParentAssociation and ShiftParentClassHelper form a loop that traverses the whole inheritance chain. The external causalities defined between them (Fig. 6.12) imply that they can be composed only one way (Fig. 6.13). Therefore, with respect to termination properties, their composition behaves as an exhaustively executed rule. The parameters defined between the rules ShiftParentClassHelper and AddParentAssociation on the backward edge of the loop (Fig. 6.14) defines the exact E-based composition of the composed metamodel-based model transformation rules (ShiftParentClassHelper \( \ast_E \) AddParentAssociation).

Proposition 6.51. If there is no directed loop in the inheritance chain the exhaustively applied transformation rule \( S_C \) depicted in Fig. 6.13 and specified by the external causalities presented in Fig. 6.14 always terminates for finite input models.

Proof. An exhaustive transformation rule is executed continuously as long as LHS of the rule can be matched to the input model. Composing the transformation rule \( S_C \) with itself results the following. The external causalities specified between the rules ShiftParentClassHelper and AddParentAssociation on the backward edge (Fig. 6.14) define that the node ParentClass2 from

<table>
<thead>
<tr>
<th>Name</th>
<th>Start Node</th>
<th>Destination Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>parentClass</td>
<td>ParentClass (2253)</td>
<td>ParentClass (2059)</td>
</tr>
<tr>
<td>class</td>
<td>Class (2251)</td>
<td>Class (2251)</td>
</tr>
<tr>
<td>classHelper</td>
<td>ParentClassHelperNode (2254)</td>
<td>ParentClassHelperNode (2250)</td>
</tr>
</tbody>
</table>

Fig. 6.14. External causalities between rules ShiftParentClassHelper and AddParentAssociation
Chapter 6. Control Flow Related Properties of Model Transformations

rule ShiftParentClassHelper is mapped to the node ParentClass of the rule AddParentAssociation. This means that while the rule $S_C$ is composed with itself, the node ParentClass2 from RHS must be mapped to the node ParentClass from LHS. Therefore, at each composition LHS of the rule increases with a new element of the traversed inheritance chain (ParentClass2), this means that there is no finite upper bound of LHS. Composing the rule with itself implies that the cumulative size series of LHS exceeds all limits. Based on termination criterion proven in [147] the rule $S_C$ terminates.

Proposition 6.52. For finite input model $G_0$ the transformation Class2RDBMS presented in Fig. 6.1 terminates.

Proof. The loop formed by the transformation rules AddParentAssociation and ShiftParentClassHelper is proven that terminates (Proposition 6.51). The rule CreateTable is a TC rule, therefore the large loop on the top terminates. The rule ProcessAssociation is also a TC rule, thus it terminates (Proposition 6.47). The last three rules remove the helper and temporary elements, the input model is finite, therefore they also terminate. Each part of the transformation terminates that means the whole transformation terminates.

6.7 Chapter Summary

This chapter has provided a control flow technique for model transformations based on graph rewriting. The transformations are represented in the form of explicitly sequenced transformation rules.

As it was presented, a control structure language needs a sequence as well as a conditional branch mechanism, hierarchy, parallel executions, and iteration constructs. VCFL has all these control structures in a deterministic implementation.

Termination is an important issue for model transformations. Since model transformations can become very complex, not only the application of single transformation rules is considered, but also transformations where rule applications are restricted according to a strict control flow.

In this chapter, the properties of the VMTS Visual Control Flow Language have been discussed. Several termination criteria for transformations have been stated and proven. An algorithm to validate the termination has also been provided.

The discussed method is independent from the input models, based on the transformation rules and the control flow model it makes its decision, therefore, the result of a transformation examination holds for all input models. For example if the termination algorithm states that a transformation terminates, then it holds for all input model that corresponds to the input metamodel.

The main limitation of the presented approach is that the termination in many cases cannot be decided based on the VCFL transformation model. Since the termination, in general, is undecidable [166], it is expected that there are several cases when the algorithm cannot make a decision.

Using the presented metamodel-based model transformation rule composing method, the behavior of a transformation is predictable. Not only its termination, as it is presented in
Section 6.5, but also other properties. For example, the method can be used to examine whether a transformation preserve any property of the input model, such as an attribute value or a relation between model nodes.

In [23] a theory is developed for the DPO approach. It provides an abstract termination criteria by a measure function $F$, and among other conditions it must hold that $F(L) > F(R)$ for a production $p$. The paper also shows concrete termination criteria such as the number of nodes, the number of edges. However, this criteria is violated in the presented case study in both rules with respect to the concrete criteria of edge and node numbers.

The layering approach is not applicable for VCFL transformations. There are situations, where its criteria do not hold. In the case study rule $ShiftParentClassHelper$ creates an element ($dstParentHelper$), and deletes an element of its type. In CLGGs [53], there is no strict control flow, rules are created without any fixed order and assigned to different layers. In VCFL, a fixed control flow is specified by stereotyped activity diagram. Therefore, the termination conditions of fixed control flows with given transformation rules and rule structures without any rule order modification must be examined.

Transformation rule composition is an important part of the presented approach. Unfortunately, it cannot be applied always, because, in certain cases, there is no possible mapping between rules. Moreover, sometimes there are a huge number of the possible compositions that all must be examined. In VMTS, the control structures are as strict as possible, and nondeterminism is avoided if possible. Moreover, external causalities decrease the number of the possible $E$ graphs, since the nodes and edges connected by a morphism from $S_i^{RHS}$ to $S_{i+1}^{LHS}$ must be mapped to the same edges in $E$ by $e_1$.

Another contribution is that in the composition of the transformation rules not only metatypes of the nodes but the external and internal causalities specified between the nodes have also been considered.

The introduced approach can be generalized to other control flow languages that facilitate to assign constraints to transformation rules and supports constraint evaluation.

VCFL has successfully been applied in industrial projects, the details can be found in Chapter 8.
Chapter 7

A Synopsis of Model Transformation Systems

Model transformation problems can be formulated as graph transformation problems, thus, a variety of tools choose this technique as the underlying mechanism for the transformation engine. Many approaches have been introduced in the field of graph grammars and transformations to capture graph domains. These approaches are specific to the particular system, and each of them has some features that others do not offer. In this chapter, a synopsis is given on model transformation systems concentrating on their control flow, attribute transformation and constraint management support. Section 7.1 introduces the model transformation approaches closely related to VMTS and compares their features with VMTS constructs. Section 7.2 provides more comparison information from the point of certain model transformation features view. Furthermore, in Appendix C, Table C.1 also gives a comparison on control flow, constraint, and attribute transformation support of these model transformation tools.

7.1 Model Transformation Approaches

The model transformation engine, the control flow support and the constraint validation methods of VMTS benefit from the results of the mathematical background of formal languages, graph rewriting, aspect-oriented software development and research related to the metamodel-based software model transformation. It also incorporates several ideas from the environments introduced here.

7.1.1 GReAT

The Graph Rewriting and Transformation (GReAT) framework [2] [3] [95] is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts. The control structure of the GReAT allows specifying an initial context for the matching to reduce the complexity of the general matching case. The pattern matcher returns all the possible matches to avoid the inherent non-determinism in the matching process. The execution engine
chooses a path nondeterministically, and the path that is chosen is executed completely before the next path is selected. The attribute transformation is specified by a proprietary attribute mapping language, whose syntax is close to C. LHS of the rules can contain OCL constraint to refine the structure but postconditions are not supported. GReAT does not address the issue of transactions, as all rule execution is assumed single-threaded.

The parameter passing and rule nesting functions are similar in GReAT and in VMTS. Both environment use and OCL-based transformation rule refinement. GReAT applies an interpretation-driven constraint evaluation method that increases the complexity of the transformation execution. VMTS has an OCL compiler that compiles OCL constraints into source code and a binary file that is capable to evaluate OCL constraint during metamodel instantiation and model transformation [123]. Both environment supports iteration and recursion in the control flow, but while GReAT applies a test rule-based branching construction, in VMTS a more efficient and user-friendly constraint-driven branching mechanism is provided. The constraint-driven branching mechanism of the VMTS is unique: it takes into consideration the actual state of the input model and as well as the result of the last rule firing (Section 6.2.2).

GReAT and PROGRES have a test rule construction. A test rule is a special expression and it is used to change the control flow during execution. A test rule has only LHS. Test rules raise the number of the rules in a control flow model, increase the complexity of the execution and make the management more difficult. If a test rule is successful (matching was successful), the rule after the test node is executable.

7.1.2 PROGRES

PROgrammed GRaph REplacement System (PROGRES) [56] [171] [175] [201] 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 provides constructs for rule firing and for sequencing the rules to form a controllable transformation process. Both imperative and declarative approaches can be used in either a deterministic or a nondeterministic manner. Designing the control structures of PROGRES, one had to take care to preserve the main properties of tests and graph rewriting rules on the level of transactions. These properties are the followings.

(i) The \textit{atomic} character: the application of a simple production either replaces one subgraph by another one and terminates successfully or fails without causing any graph changes. As a consequence of this fact, the execution of a sequence of graph rewriting rules should also either terminate successfully or fail without any effects on the graph. (ii) The \textit{boolean} character: simple productions and complex transactions signal their state of termination to the calling environment. This implicit (boolean) return value determines the selection of the next graph rewriting rule. (iii) The \textit{nondeterministic} character: Even in simple cases there may be more than one subgraph of the current graph which is isomorphic to the left-hand side of the applied production.

PROGRES can also specify static integrity constraints on the graphs. This is done with a language called \textit{schemas} that define the graph domain. However, PROGRES is not suitable for specifying model-interpreters because: (i) schemas are not as powerful and as widely used as UML class diagrams to specify integrity constraints, (ii) PROGRES deals mainly with trans-
formations on a single graph and does not produce a new graph that conforms to a different metamodel (schema), and (iii) PROGRES is mainly a programming language with graphical productions and, consequently, not at the level of abstraction desired for specifying model-interpreters [22] [174].

7.1.3 VIATRA

VIATRA (Visual Automated Transformations) [34] [196] [197] [198] [199] is a model transformation framework developed mainly for the formal dependability analysis of UML models. In VIATRA, metamodeling is conceived specially: the instantiation is based on mathematical formalisms and called Visual Precise Metamodeling. The transformation language of VIATRA supports type checking, negative patterns, attribute conditions and traditional pattern matching issues.

VIATRA2 is a recently implemented tool, it is an Eclipse-based [26] general-purpose model transformation engineering framework. It supports the entire life-cycle for the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains.

VIATRA2 is able to cooperate with an arbitrary external system, and execute the transformation with a native transformation model (plug-in). The rule specification language is a proprietary pattern language with type information. The attribute specification is graph-based. The attribute transformation is performed by abstract state machine statements, and there is built-in support for attributes of basic Java types. The model 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.

VIATRA and AGG applies NACs. Using NACs it is possible to express structures and attribute value, but multiplicity conditions cannot be defined without additional constraints. In VMTS multiplicities can be expressed on metamodel-based model transformation rules.

7.1.4 AGG

The Attributed Graph Grammar System (AGG) [37] [90] [191] [192] is a rule based visual language supporting an algebraic approach to graph transformation. It aims at the specification and prototypical implementation of applications with complex graph-structured data. AGG may be used (implicitly in "code") as a general purpose graph transformation engine in high-level JAVA applications employing graph transformation methods. AGG graphs may be attributed by Java objects and types. Basic data types as well as object classes already available in Java class libraries may be used. The graph rules may be attributed by Java expressions which are evaluated during rule applications. Additionally, rules may have attribute conditions being boolean Java expressions. Similarly to GReAT, this means online interpretation of the conditions.

In AGG, termination criteria are implemented for Layered Graph Transformation Systems (LGTS). The proposed criteria are based on assigning a layer to each rule, node and edge type. For termination, they define layered graph grammars with deletion and non-deletion layers. Termination criteria are expressed by deletion and non-deletion layer conditions. The layers fix the order how rules are applied. The interpretation process first has to apply all rules of layer
0 as long as possible, and then all rules of layer 1, etc. Rule layers allow specifying a simple control flow graph transformation. Once the highest layer has been finished the transformation stops, unless the option "loop over layers" is turned on.

The layer-based examination of the termination (both in case of the AGG and AToM\(^3\)) cannot be applied in general because, there are cases when the layering conditions do not hold for certain transformation rules (e.g. for the rule \textit{ShiftParentClassHelper} in the case study \textit{Class2RDBMS}).

### 7.1.5 AToM\(^3\)

The transformation and simulation tool AToM\(^3\) (A Tool for Multi-formalism and Meta-Modelling) \cite{37} \cite{38} \cite{39} uses model transformation to simulation traces in order to simulate the operations. The rule constraints can contain generalized negative application conditions and can be pre- and postconditions to events. Constraints can be both semantic and graphical constraints. Similarly to AGG, the control flow consists of layers; the rules are sequenced by priority numbers within the layers. A rule is executed only once, but in case of non-overlapping matches, the rules are applied to all the matches.

### 7.1.6 FUJABA

In FUJABA (From UML to Java and Back Again) \cite{88} \cite{99}, the combination of activity diagrams and collaboration diagrams are used to express control structures. The story-diagrams are a visual programming language that facilitates the specification of complex application-specific object structures. Moreover, FUJABA has extended story-diagrams by statecharts to story-charts. Story-charts use statecharts and activity diagrams to define complex control flows and collaboration diagrams to specify the entry, exit, do, and transition actions that deal with complex object structures.

The control flow support of the FUJABA and VMTS are similar, but FUJABA supports only UML diagrams and Java source code generation. Furthermore, in FUJABA the validation method is based on testing.

### 7.1.7 OMG QVT

The Model-Driven Architecture offers a standard interface to implement model transformation tools. The transformation related part of MDA is the Query, Views, Transformation for MOF 2.0 \cite{73} \cite{161} \cite{180}. Three types of operations are provided: queries on models, views on metamodels and transformation on models.

In this terminology a \textit{query} is an expression that is evaluated over a model. It cannot change the model, it is fully declarative. Queries can be constructed using UML Action Semantics as well as OCL \cite{182}. They can be regarded parallel to XPath in XML. A \textit{view} is a model derived from another model (base model) via transformation. Any change in the view or in the base model affects the other. The metamodel of the view can be different from that of the base model. Views can be readonly. A query can be considered a restricted view.
A transformation generates a target model from the source model. Transformations can lead to either independent or dependent models. The difference is that in the first case there is no relationship between the target and the source model after the transformation process. The units of the transformation are called rules. A simple transformation transforms single elements in the source model to single elements in the target model. Quite often, the source and target models have essentially the same structure. A complex transformation builds structures in the target model which do not directly correspond to any individual element in the source model.

The semantics of the QVT Core and Relations languages allow for the following execution scenarios: (i) Check-only transformations to verify that models are related in a specified way. (ii) Single direction transformations. (iii) Bi-directional transformations. (iv) The ability to establish relationships between pre-existing models. (v) Incremental updates when a related model is changed after an initial execution. (vi) The ability to create as well as delete objects and values, while also being able to specify which objects and values must not be modified.

Bi-directional transformations are only possible if an inverse operational implementation is provided separately. In an inverse transformation executed in VMTS, the modification, creation, and deletion of the model structures can be achieved with swapping LHS and RHS graphs and inverting the modifications described by internal causalities. But the attribute modification is hard in the case of the QVT as well. In general, the generated target model does not contain all information covered by the source model. Therefore, it is not possible to correctly produce the source model from the target model with an inverse transformation. In summary, general inverse transformation should contain totally new internal causalities that perform attribute modifications.

VMTS trace objects are created based on the internal causalities of the transformation steps. Using the trace objects with the help of constraint management VMTS supports the model evolution. Once a relationship has been established between models by executing a transformation and creating trace objects, changes to a source model may be propagated to a target model by re-executing the transformation in the context of the trace, causing only the relevant target model elements to be changed, without modifying the rest of the model.

QVT does not have a visual control flow support. Furthermore, the branching mechanism provided by the when-where clauses is a bit difficult to use. Often, model-to-model and model-to-code transformations need to have a strict control over the execution sequence of the rules, therefore, VCFL models provide a more comfortable way to build transformations from individual rules.

In summary, QVT can be realized by graph rewriting-based model transformation, therefore QVT can utilize the results originating from the formal background of graph transformation.

### 7.1.8 Other Approaches

In [24] an approach is presented that is based on a transformational technique that has been developed and applied to obtain system architectures from requirements specified as UML use cases. It is presented that such a technique can be applied to product lines. For presentation purposes the GoPhone product line that uses the UML modeling language is chosen.
### 7.2 Comparison of Model Transformation Approaches

**Typing Information.** In all the considered approaches, the typing information is given by an attributed type graph or metamodel which contains the structural information (typing entities and attributes), inheritance concepts and multiplicity constraints. In AToM³, additional constraints can be expressed in Python, while AGG and VIATRA allow the formulation of graph constraints. In QVT, typing information is provided from the source and target metamodels for model transformation.

**Transformation rules.** In general, a graph transformation rule is defined by a rule application to the instance or input models. Rules consist of a left-hand side (LHS), a right-hand side (RHS), negative application conditions (NACs) and the corresponding graph morphisms. In QVT the relationship between source and target elements is defined by relations which could be compared to rules in the graph transformation approach. In VMTS LHS and RHS graphs can be built from the elements of not only one but several metamodels.

**Preconditions.** The main precondition for transformation rule application is LHS of the rule. Furthermore, additional application conditions could be added. Rules also have additional attribute conditions which are expressed in Java, Python, or OCL. VIATRA2 and AToM³ support graph pattern with arbitrary levels of negation, furthermore, AGG and VIATRA2 use graph conditions: NACs and positive application conditions. Moreover, in some approaches, the gluing condition is evaluated when the rule is going to be applied. In QVT pre-dependencies of a relation are defined in a postcondition starting with the keyword when. In VMTS metamodel-based rules facilitates to assign OCL constraints to them.

**Postconditions.** Application conditions which are checked after a rule execution are supported by the approaches in a similar form as preconditions. But the semantics can differ: While in AToM³ the condition is checked only, AGG, VIATRA2 and VMTS provide a rollback mechanism if the condition is not met. In QVT, post-dependencies of a relation are defined in the postcondition starting with the keyword where.

**Actions.** The actions of a transformation rule can comprise deletion and creation of model elements as well as attribute modification. The latter is specified in Java, Python, C and XSLT, dependent on the approach. In addition, pre- and postactions to be executed before and after the rule application can be specified in AToM³.

**Control.** Uncontrolled transformation rule application results in nondeterministic rule selection. To increase the efficiency of graph transformation, a variety of control concepts for rule and match selection have been considered. Some of them are used with the approaches presented: PROGRES has transactions, AGG uses rule layers, AToM³ supports priorities for rules, VIATRA2 supports ASMs, FUJABA applies story diagrams, and VMTS offers stereotyped activity diagrams. Both GReAT and PROGRES support deterministic and nondeterministic rule selection. Parameter passing, iteration, and recursion are supported by GReAT and VMTS. Branching is solved by test rules in PROGRES and GReAT, while VMTS offers constraint-driven path selection. In QVT, top relations are applied directly whereas relations are called from top relations. Furthermore, pre- and postactions to be executed before and after the control can be defined in VMTS.
Graph grammar techniques such as node replacement grammars, hyperedge replacement grammars, and algebraic approaches such as the ones used in AGG do not provide sufficiently expressive mechanisms for controlling the application of transformation rules. PROGRES has a rich set of control mechanisms; however, they only perform transformations within the same schema.

**Validation.** AGG, VIATRA2 and FUJABA support some validation tools which check if constraints are preserved by models. Moreover, type checking is supported. Furthermore, AGG offers validation techniques to find conflicts between rules and to check termination criteria. AToM³ offers a code generator for AGG to use AGG’s validation tools within AToM³. VMTS provides the following validation mechanisms: (i) validation of the models based on their metamodels, (ii) validation of the control flow specifications (isolated, illegal rules, or nondeterministic branching possibilities), and (iii) online validation of the model transformations.

### 7.3 Chapter Summary

Compared to other approaches, VMTS meets the expectations in model-to-model and model-to-code transformation. VMTS has state of the art mechanisms for validated model transformation, constraint management and control flow definition. The environment has several standalone algorithms and other solutions that makes them efficient.

VMTS has a unique constraint management support, to our knowledge, no another environment supports aspect-oriented constraint management with different constraint propagation methods.

VMTS provides a high-level control flow language with several constructs that optimize and make the transformations highly configurable: external causalities, efficient branch selecting, and pivot nodes. The constraint-driven branching mechanism of the VMTS is unique in the sense that the decision is made not only based on the actual state of the input model but using system variables (SystemLastRuleSucceed, SystemLHSFailure and SystemRHSFailure) as well. If a transformation rule fails and the next element in the control flow is a decision object, then it could provide the next branch based on the constraints. This VMTS construct accelerates and makes the transformation more efficient and the control flow model simpler, because there is no need to define test rules. Furthermore, the approach supports the examination of the control flow related properties such as termination.

The plug-in-based visual user interface of VMTS makes the constraint management and transformation specification easy and flexible.
Chapter 8

Application of the Results

The results - general validation, preservation and guarantee, aspect-oriented constraint management, controlled visual model transformation, and the examination of transformation properties - presented in Chapter 4, 5 and 6 - together they form a validated efficient visual model transformation. These techniques are successfully applied in industrial applications like source code generation from statechart, class diagrams and resource model [114] [118] [131] [141] [156], furthermore, model-based unification of mobile platforms [68] [69] [123] [141] [142], and model-to-model transformations to support model-based development [113] [120] [193].

In this chapter, an overview is given on the basic concepts of Visual Modeling and Transformation System. That is followed by a detailed description of the constraint-driven model transformation capabilities of VMTS. It includes the transformation rule and the control flow specification, and the constraint management. Finally, three case studies offering solutions to practical problems are provided.

8.1 The Visual Modeling and Transformation System

Metamodeling and graph transformation techniques have successfully been used in software system modeling. The advantages of these two methods have been realized by Visual Modeling and Transformation System (VMTS) [91] [111] [114] [143] [145] [146]. In this section, the concepts of an n-layer modeling and metamodel-based model transformation framework are outlined.

VMTS benefits from the results of the mathematical background of formal languages, graph rewriting and results related to the metamodel-based software model transformation. VMTS is an approach that uniformly treats model storage and model transformation. This facilitates the notion of the metamodel that also makes VMTS a highly configurable tool.

VMTS has an efficient multilayer metamodeling solution. The metamodeling system uses only one instantiation method between the instantiation layers that results the layer transparency. This means that each layer is handled by the same functions within the tool. VMTS supports UML 2.0 diagrams, feature modeling and other domain-specific languages [91]. Furthermore, it has powerful constructs for model transformation applications providing well-defined constructs for transformation rule specification, attribute transformation, and control flow. The
Chapter 8. Application of the Results

Fig. 8.1. VMTS block diagram

The basic formalism used by VMTS is labeled directed graphs. Attributes are represented in XML, and attribute transformations are XSLT scripts. In a model transformation system, the control flow support is a highly required issue. In VMTS, rules are based on UML class diagrams and the standard instantiation relationship between UML class and object diagrams. VMTS Visual Control Flow Language (VCFL) (Chapter 6) is specified as a stereotyped UML activity diagram with external causality and constraint-driven branching support.

VMTS Rule Editor, VMTS Control Flow Designer and VMTS Constraint Manager along with VMTS AO Constraint Manager and VMTS Constraint Weaver are proof-of-concept implementations of the theoretical results presented in this thesis. Efficient aspect-oriented constraint management, well-defined control flow and validated model transformation facilitate the practical applicability of the approach for model-to-model and model-to-code transformations. Fig. 8.1 depicts the block diagram of VMTS.

VMTS Presentation Framework (VPF) uses a flexible plug-in-based architecture to offer individual, metamodel-dependent visualization and editing features [91]. The user interface (Adaptive Modeler) is functionally separated from the model storage unit (AGSI Core, Attributed Graph Architecture Supporting Inheritance), which uses a Relational Database Management System (RDBMS) to store the model information. AGSI handles graphs with the constructs of nodes, directed edges and labels assigned to nodes and edges. The labels are XML files,
Fig. 8.2. Transformation rule CreateParentClassHelper

and the attributes are represented in an XMI [162] format. The model transformation can be accomplished by traversing model processors (TMPs) or visual model processors (VMPs).

The traversal approach is the simplest way of the model processing. TMPs traverse the input models and change the appropriate parts of them or produce an output model. Model processing code generators are special TMPs.

8.1.1 Visual Model Processing in VMTS

Visual Model Processors (VMPs) provide a visual techniques for model transformation. In VMTS, graph rewriting is the underlying transformation technique. Graph rewriting rules are based on the double pushout approach (Section 3.1).

In Fig. 2.3, the principles of VMTS metamodel-based model transformation are depicted with its input and output artifacts. A detailed description about the model transformation process including the validation of the pre- and postcondition can be found in Section 4.2.3 and in Fig. 4.2.

Based on the theoretical results presented in previous chapters this section introduces the method of the transformation rule and control flow model specification in VMTS. Furthermore, the constraint management capabilities of VMTS are also presented.

8.1.1.1 Specification of Transformation Rules

In VMTS, the transformation rule specification is achieved by VMTS Rule Editor that is a plug-in integrated into VPF. The metamodel of the VMTS Rule Editor is depicted in Fig. 6.2b. An example transformation rule that creates a ParentClassHelperNode is presented in Fig. 8.2.

The meta attributes of the transformation rule are given in Fig. 8.3. In VMTS, LHS and RHS are defined based on UML class diagram syntax. LHS and RHS of the transformation rules are built from metamodel elements. An LHS or RHS can have any number of metamodels. It is necessary, because during the transformations, it is often required that nodes from different models can be connected by helper nodes. Therefore, each node appearing in LHS or RHS can have a metatype from any metamodels. LHS is the metamodel of the matched submodel and
RHS is the metamodel of the generated model. Therefore, during the transformation process, a part of the input model must be found which instantiates LHS, and the generated model is an instance model of RHS. The constructs that form a metamodel-based LHS specification are inheritance and multiplicity support. Inheritance support is analogous to the type compatibility of object-oriented languages. The inherited class can always 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.

Causalities defined between LHS and RHS elements can express the modification or removal of an LHS element, and the creation of an RHS element. XSLT scripts can access the attributes of the object matched to LHS elements, and produce RHS elements to which the causality points. Therefore, it is not true that if a rule element appearing in LHS but not in RHS is going to be deleted.

When a transformation rule is executed, the internal causalities are processed. Firstly, causalities with deletion type are performed. Secondly, creations are executed. The metatype of the new element is retrieved from RHS, their attributes are generated by XSLT scripts, and the connections are established based on RHS structure. Finally, modification causalities are processed, which means XSLT script execution on the attributes of the element matched to LHS node.

The metamodel-based specification of the transformation rules allows assigning OCL constraints to the PRNs and edges, using the guidelines of the UML standard. These constraints are bound to the transformation rules, thus they are able to express local constraints. Elements not appearing in LHS or RHS cannot be included in the OCL expressions. Although the specification has this local nature, it does not mean that validating them ignores checking other model elements in the input model.

---

<table>
<thead>
<tr>
<th>Instance</th>
<th>Common</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InstanceName</td>
<td>TransformationStep</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>IntraCausality</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>RHSMetamodel</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>LHS-Metamodel</td>
</tr>
<tr>
<td>TypeExpression</td>
<td>existing</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>1</td>
</tr>
<tr>
<td>Editor</td>
<td>Vmms-FujiEditor-Plugin-MetamodelSelector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance</th>
<th>Common</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InstanceName</td>
<td>PatternRuleNode</td>
<td></td>
</tr>
<tr>
<td>[A]</td>
<td>Name</td>
<td>MetaType</td>
</tr>
<tr>
<td>[A]</td>
<td>TypeExpression</td>
<td>existing</td>
</tr>
<tr>
<td>[A]</td>
<td>Multiplicity</td>
<td>1</td>
</tr>
<tr>
<td>Editor</td>
<td>Vmms-FujiEditor-Plugin-MetamodelEditor</td>
<td></td>
</tr>
<tr>
<td>PathSelection</td>
<td>Draw</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 8.3.** Transformation rule and pattern rule node attributes
8.1.1.2 VMTS Control Flow Support

A completely uncontrolled transformation rule application allows nondeterministic rule and match selection at any rule of the transformation. To increase the usability of model transformation control, concepts for rule selection must be applied. There is a need for a high-level control flow language that can control the application of the rules and allow the modeler to manage the complexity of the transformation.

Recall that the VMTS control flow models are defined by stereotyped UML activity diagrams, where the activities are provided with the stereotype \textit{<< Transformation Rule >>}. The metamodel of VCFL is presented in Fig. 6.2a. An example VMTS visual control flow (VCFL) model is given in Fig. 6.1.

The decision objects contain OCL constraints to decide which branch must be passed the control. The OCL expressions may contain system variables, such as \textit{SystemLastRuleSucceed}, which is true if the last rule has been executed successfully, false otherwise. VCFL allows specifying external causalities, which are a means of parameter passing between the rules. External causalities can be defined between RHS of a rule $S_1$ and LHS of a rule $S_2$, where there must exist a transition from $S_1$ to $S_2$ in the control flow. If the control is passed to this transition, the model objects matched to the source of the causality are automatically associated with the target of the external causality. The matching algorithm considers these LHS objects already bounded.

As it was mentioned in Section 6.5, a VMTS transformation rule has two specific attributes: \textit{Exhaustive} and \textit{MultipleMatch}. The difference between them is the following: an exhaustive transformation rule is executed continuously as long as LHS of the rule could be matched to the input model. The \textit{MultipleMatch} attribute of a rule facilitates that all occurrence of LHS must be found and replaced in the input model.

An optimization method is to start the transformation with an initial binding. This binding gives an initial match to the first rule of the transformation. In VMTS, this construct is referred to as pivoted transformation. A pivot node is an input parameter of the control flow specified by the user that exactly determines the starting point of the matching. Pivoted control flow achieves a significant saving, because in this case only a part of the input model has to be taken into account.

8.1.1.3 Constraint Management

Several model transformation approaches exist which facilitates executing transformation rules. Many of the tools also offer control flow support. However, often there is only little or difficult support for validating any properties of input or output models.

In VMTS, LHS constraints are considered as preconditions of the transformation rule, and RHS constraints postcondition of the rule (Section 4.2.2.1). They provide the possibility for verifying the correctness of the matched part of the input model online, and the generated output model.

VMTS provides three different ways for specifying constraints:
In VMTS each node and edge may has any number of constraints, this is supported by the framework. VMTS Constraint Editor facilitates to define and propagate constraints to individual PRNs or edges.

VMTS Aspect-Oriented Constraint Editor also supports the constraint specification. In this case constraints are stored independently from the transformation rules. They can be used during weaving methods. AO constraints are identified by their name but they can be filtered also by their context information and the metamodel of their context type.

Constraint aspects are defined using a VPF plug-in (VMTSConstraintAspectPlugin). They are stored as individual models with their structure, OCL constraints, and weaving constraints. Constraint aspects can be normalized automatically after their creation and modification, or later when the modeler requires it. They can be woven optional times to different transformations using Constraint Aspect Weaver (Section 5.4.1.2).

In the second and third cases the constraints are stored as aspects and can be propagated later to their appropriate places. The result of the constraint propagation is a weaving configuration that connects the transformation rules and the constraints.

Because of the different constraint notations and requirements VMTS gives four different methods to propagate constraints to the transformation rules:

- **VMTS Constraint Editor** can be used to specify OCL constraint and directly propagate them to PRNs and edges.

- **VMTS Base Constraint Weaver** facilitates to propagate aspect-oriented constraints to a set of PRNs and edges. PRNs and edges must be selected from a list one-by-one. The selection is supported by transformation and transformation rule-based filtering possibilities. The required AO constraint is linked to all the selected elements by a button pressing.

- **VMTS Global Constraint Weaver** makes possible to propagate AO constraints to whole transformations. The required constraint type (Validate, Preserve, Guarantee, Precondition, Postcondition) must be given by the modeler. Based on the type, GCW (Algorithm 5.5) determines the PRNs and the edges of the transformation rules where the AO constraint (or the automatically created constraint pair) must be propagated. The result of the weaving, the weaving configuration including the woven items, is shown after the propagation, but it also can be checked later.

- **VMTS Constraint Aspect Weaver** works similarly to GCW with the difference that it weaves constraint aspects instead of AO constraints.

VMTS supports the weaving processes and transformation execution with easy-to-use user interfaces.
8.2 Model-Based Development with VMTS

In this section, case studies are provided to introduce the applicability of the presented methods. The first case study is an MDA-based model transformation, where source code is generated for different mobile platforms from the same input models. The second case study is a model-to-model transformation that generates database representation from class diagrams and emphasizes the importance of aspect-oriented constraint management. Finally, the third one is a model-to-code transformation that has been chosen to show how the approach supports software evolution.

8.2.1 Model-Based Unification of Mobile Platforms

With the introduction and popularity of wireless devices, the diversity of the platforms has also been increased. Developing software for different mobile devices requires more and more time and work investment because of the incompatibility of mobile platforms. Creating a common development platform requires a higher abstraction level. This case study, starting from the fact that the software development for incompatible mobile platforms is problematic, provides an MDA and domain-specific modeling-based solution.

The platform commonalities and diversities can be described with a domain-specific modeling language (DSML) and the final products can be automatically generated from these models. This method is supported by the domain-specific model processors that are specifically implemented to support the domain concepts found in the models. DSMLs allow developers to specify their solutions on a higher level of abstraction. Model processors automatically generate the lower level artifacts. The approach improves and accelerates the software or system development process.

Incompatible platforms require implementing the same functionality separately for each different platform. This issue has been solved by unifying the platforms with MDA-based visual model transformation. In VMTS, the key of the platform-independence is the visually defined model transformation.

The case study introduces how VMTS generates source code for different mobile platforms from the same resource model and a statechart diagram, applying transformation methods. The selected mobile platforms are the Windows Mobile [71] [179] [194] and the Symbian [165] platforms. The goal of this method is to reuse the platform-independent software models and generate the platform-specific source code with visual model processors. If the statechart diagram of the case study is specified in detail, the generated code will handle the user interface described by the resource model on both platforms.

Fig. 8.4 depicts the block diagram of the case study that is a mobile application which can be used to order cinema tickets with cellular phones. The application facilitates to select an appropriate cinema, a movie with its start time, the number of the tickets and to send the order. The input models are (i) a resource model which describes the user interface of the Cinema Ticket application, and (ii) a statechart diagram which represents the required operation of the user interface described by the resource model. The input models are prepared with VMTS Adaptive Modeler, using the VMTS Resource and Statechart modeling plug-ins.
The resource model created according to the resource metamodel contains a Form with the necessary controls (Page, ButtonBar, Menu, MenuItem, RadioButtonList, and Slider). The statechart diagram defines how the application should process the events generated by the user. The transitions between the states represent the events and the states together with their actions, they describe the required behavior.

Different VMTS VMPs are applied for different target platforms. The input models of each VMP are the same models, but the results of the transformations, the generated artifacts, are platform-specific models.

From the statechart diagram, VMTS generates a CodeDOM tree [194] for each event handler. The CodeDOM tree is a language-independent model representation of the source code. This means that the code generation is a syntax tree composition, from which the .NET Framework [194] generates the source code using the System.CodeDOM namespace. The resource model is processed differently for the various platforms. In case of the Symbian branch the resource model is transformed into another model which represents an XML file and contains all information related to the user interface. The source code generated from the statechart model is merged into the XML file. This XML file is the input of the XML2C [91] application that is used to generate the C++ source code for Symbian platform. The generated source code uses the Simplian Class Directory [1] [103] and runs on a Symbian OS-based cellular phone. The XML2C is part of the Simplian framework that makes the development easier for Symbian platform. For Windows Mobile platform, the resource model is also converted to CodeDOM model, and the two CodeDOM models (generated from statechart and resource models) are joined. The source code for the Windows platform is generated from this model.
The transformation rules \textit{Resource2XML} and \textit{Statechart2CodeDOM} are presented in Fig. 8.5. The constraints assigned to the PRNs require the rules to match only those controls and states which do not have a generated XML model or CodeDOM tree.

\begin{verbatim}
context Control inv generated_XML:
  not HasGeneratedXML

context State inv generated_CodeDOMTree:
  not HasGeneratedCodeDOMTree
\end{verbatim}

After matching LHS successfully, rules are fired according to the internal causalities, and finally, the postconditions are evaluated on the result of the transformation rules. The postconditions of the transformation rules check whether the \textit{HasGeneratedXML} property of the controls and \textit{HasGeneratedCodeDOMTree} property of the states affected by LHS part of the rule is true. A transformation is successful if all the postconditions hold for the generated result: the generated output satisfies the required conditions. This means that in case of success the output is valid [119] [123] [141].

8.2.2 Model-to-Model Transformation with Strictly Controlled Model Transformation

This section completes the already introduced facts of the transformation \textit{Class2RDBMS} that generates database model from class diagram.

In general, there are two approaches to database design: attribute-driven and entity-driven. Constructing class diagram is also a form of entity design. There are also two possibilities to implement the identity of the objects [20]: existence-based identity and value-based identity. The case study presented here follows the existence-based identity implementation.

A VMTS UML class model consists of classes and relations between them (inheritance, association and dependency). A class can be abstract, and it consists of \textit{ClassAttributes} and \textit{ClassOperations}. A \textit{ClassAttribute} has name, type, visibility and property-string fields. The property-string optionally shows other properties of an attribute. If one follows the value-based
identity approach the property-string field can be used to mark attributes as primary key: \{primary_key = 'true'\}. An example class diagram is depicted in Fig. 8.6 that will be the input model of the case study.

The metamodel for RDBMS models along with its instance, and the required output of the case study are presented in Fig. 8.7. An RDBMS model consists of one or more tables. A table consists of one or more columns, which are defined as attributes of the metatype Table. The attributes of the tables and relations describe the following. One of the columns will be primary key. A table may also contain zero or more foreign keys. Each foreign key refers to the particular table that it identifies, and denotes one column in the table as being part of the foreign key.
In order to transform a class model to ideal tables we must choose among several mapping alternatives. For example there are several ways to map inheritance relations or associations to RDBMS models.

The requirements stated against the transformation Class2RDBMS are presented in Section 2.4.

The database metamodel is supported by VTMS Database Plugin integrated into VMTS Presentation Framework. The custom editors offered by the database plug-in facilitates to edit columns and indexes of the tables and to specify table relations.

The VCFL control flow model of the transformation is depicted in Fig. 6.1. In Section 4.3, the transformation rule CreateTable of the case study is presented with its pre- and postconditions. The constraints facilitate to require certain properties form the transformation rule and make it validated. Furthermore, the constraint management of the case study is presented in Section 5.6. The aspect-oriented approach (Chapter 5) facilitates the consistent and simple management of the constraints [120] [130]. Finally, in Section 6.6 the termination properties of the case study have been discussed.

Processing the association in the case study can be optimized with new constraints. The matching process can differentiate the edges based on their allowed multiplicities using constraints. This solution facilitates that transformation rules along with their XSLT scripts become simpler, which makes debugging the transformation and searching errors more efficient.

8.2.3 Model Transformation-Driven Software Maintenance

Software evolution is the process of making change to software artifacts, while preserving the relationships between those artifacts. The MIC-based model transformation-driven software development process supports the following changes of the models. In addition, it facilitates code generation, helps to transform the software models into code artifacts [185].

The required method to follow the software evolution depends on the reason for the change, and not on an objective characteristic of the change. The four most frequent types of the evolution that should be differentiated are the following: specification evolution, mapping evolution, platform evolution and specification language evolution.

(i) Changes to the input model (system specification) require consequent changes to the output model or code (implementation of the system). This means that the output must be generated based on the modified and new part of the input model. This is the mostly applied evolution type. (ii) Changes to the transformation require consequent changes to the output artifacts. If the transformation rules or the control flow model is modified, then the whole output must be regenerated. (iii) Changing the platform (output metamodel) on which the system is implemented involves a new mapping and a new implementation. Using the same input, the whole output must be regenerated with the new transformation that corresponds to the modified output platform. (iv) Changing the input metamodel (language for expressing the specification) requires a new specification, new mapping and new implementation. The cases (ii), (iii) and (iv) are less frequently applied than the case (i), but they require most efforts.
The case study presented in this section introduces how VMTS follows and propagates to source code the changes made on the software models. The input and output metamodels of the case study are depicted in Fig. 8.8.

The transformation (VCFL model) of the case study contains only one transformation rule (Fig. 8.9a) built from metamodel elements: LHS from the elements of the input metamodel and RHS from the elements of the CodeDOM metamodel [91] [194]. The depicted VCFL transformation rule generates a CodeDOM tree from the input model.

The internal causalities of the rule and their XSLT scripts describe the following. For each class, the rule creates an interface of the same name, plus an inheriting implementation class.
For each attribute, it creates a private field in the corresponding class of the corresponding type with appropriate `get` and `set` methods. In addition, for each association, the transformation generates corresponding array pairs in the implementation classes, and creates `add` and `remove` methods in the interfaces. For each class, it generates a factory class for creating instances. Finally, the rule creates a system factory that links to all created factories. The result generated by the transformation is illustrated in Fig. 8.9b.

The most important question related to the modification is that how to express, manage, and propagate the changes to the output model? It would be efficient to transform only new and modified parts of the input model and not the whole again. Constraints `Const_1` and `Const_2` are used to ensure that the transformation rule generates a CodeDOM tree only from those parts of the input model that has no generated output model or that has been modified since the last execution of the transformation. In the input model each class and association have two attributes: the `HasGeneratedCodeDOMTree` and the `Modified` that indicate whether the actual input model element has a generated CodeDOM tree or it is modified. They are modified and checked by the transformation process, and the attribute `Modified` is also updated during model modification.

```plaintext
context Class inv Const_1:
    not self.HasGeneratedCodeDOMTree or self.Modified
```

Model evolution: in the input model of the case study, the type `Person` becomes abstract, and two inherited types, `Employee` and `Manager`, are added. The attribute `Modified` of the PRN `Person` becomes true, and the attribute `HasGeneratedCodeDOMTree` of the newly added PRNs is false. The modifications (Fig. 8.10) achieved on the input model are stored, and the implementation is updated based on it. In the output model (Fig. 8.11), the classes `PersonImpl` and `PersonFactory` are deleted, the `Person` is modified, and the classes `Employee`, `EmployeeImpl`, `EmployeeFactory`, `Manager`, `ManagerImpl`, `ManagerFactory` and the required associations are newly created.

As a conclusion of the case study, it can be stated that design and dependency information must be preserved for software evolution to be automated. The MDA-based approaches define design information in high-level instances, while dependency information in transformations. Therefore, the software evolution support, as it is presented, can be solved with MDA-based model transformations and refined with adequate constraint management [126].
8.3 Chapter Summary

This chapter has presented the concepts of Visual Modeling and Transformation System concentrating on its visual model processing capabilities. The fundamental structure of AGSI Core is basically a labeled directed graph. Metamodelling makes the storage system flexible and configurable. An environment built on metamodelling techniques requires only a few hard-wired constructs: node, edge, labels, inheritance, containment, and association classes. VMTS uses one type of instantiation to realize an n-layered metamodelling environment.

VMTS Presentation Framework provides a plug-in-based architecture that makes easily configurable the metamodel-based visualization. VMTS Rule Editor, Control Flow Designer and Constraint Aspect Editor are all plug-ins integrated into VPF.

Both LHS and RHS of a transformation rule can be composed of the elements of any number of metamodels. VMPs can use those metamodel elements in the transformation rule specification. Inheritance and multiplicity support makes possible to form the metamodel-based transformation rules. LHS and RHS elements are linked with internal causalities that define the behavior of the rules using XSLT scripts.

VCFL provides a high-level control flow language to define transformations. It has a constraint-driven branching construction. External causalities makes the execution of the transformation more efficient. They pass initial matches between transformation rules. Rules can be executed exhaustively or with multiple match property. Another optimization is given by pivot nodes that as input parameter of the control flow determines the starting point of the matching.

Transformation rules are on the layer of the metamodels that allows assigning OCL constraints to transformation rule elements. The results of Thesis II makes VMTS able to provide different ways for constraint specification and propagation from which the modeler can select the most appropriate for the actual task. Constraint propagation results a weaving configuration that links the transformation rules and the constraints or constraint aspects. Weaving configuration can be executed similarly to the transformations with the difference that its constraints are also taken into account during the transformation process.
Finally, three case studies have been presented in order to prove the industrial applicability of the presented methods. The first case study, based on the theoretical results of Thesis I and Thesis III, uses visually defined model transformations to realize model compilers. The input models of the transformations are the same platform-independent software models, and the resulted platform-specific models are different on each target platform. This MDA-based [70] [98] [152] [153] [181] method still involves human interaction where the expressiveness and the organization of the models are of key importance. The presented method helps the transformation designers to make their models better-organized and to conceive their model better in order to build transformation with less undiscovered error.

In the third case study it is shown that MIC-based systems support following the changes of the models and facilitates to transform the software models into code artifacts. The results provided by Thesis I and Thesis III facilitates that VMTS approach supports the software model-driven software evolution.

In addition, the presented approach has been applied to generate source code from statechart diagrams to Quantum Framework (qF) [89] [172].

Furthermore automatic code generation for visualizing domain specific languages also supported by VCFL. Although the plug-in-based architecture of VPF is flexible, it has a drawback: creating plug-ins requires manual coding. In VMTS a domain-specific language, the VMTS Presentation DSL (VPD) - that is the subject of an ongoing research with Gergely Mezei -, is used for specifying the presentation of arbitrary DSLs and the standard UML models. The VPD plug-in offers a graphical way to display and edit VPD models that specify the mapping between the metamodel elements and their presentation information. A VMTS VMP is applied to transform models defined by a VPD model automatically into source code.
Chapter 9

Conclusions

9.1 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. Furthermore, in this chapter I outline some future directions of basic research and applications.

Thesis I

Related publications: [112] [113] [118] [119] [121] [122] [123] [126] [128] [129] [131] [136] [141] [142].

I have given a constraint-driven method for validated model transformation, in addition, I have provided algorithms for efficient constraint evaluation during the model transformation.

- I have examined the relation between pre- and postconditions and OCL constraints assigned to the transformation rules.
- I have introduced the concepts of general validation, general preservation, and general guarantee.
- I have proven that if a transformation contains rules specified by high-level constructions, and the transformation has been executed successfully for an input model, then the generated output model satisfies the conditions required by the high-level constructions. Its corollary is that the successful execution of the transformation produces a valid result that is defined by the transformation rules and refined with constraints created from high-level constructions.
- I have given algorithms to support the online validation, preservation, and guarantee of certain properties during model transformation.
Chapter 9. Conclusions

- I have introduced a naive algorithm (Rule Constraint Validator) and its detailed description for constraint validation during model transformation.

- I have provided a validation algorithm (Invariant Analysis) to compare the constraints contained by the metamodel and the transformation rules immediately after transformation rule specification. Invariant Analysis decides which constraints in the transformation rule is certainly proper and which not.

  The computational time that can be saved using the Invariant Analysis is $SC_{IA} \geq n^k - C_{IA}$.

  Where $SC_{IA}$ denotes the saved computational time using IA, $n^k$ is the complexity of the matching process, where $n$ is the number of the input model nodes, and $k$ is the number of nodes in the matched submodel.

- I have suggested an algorithm (Persistent Analysis) to evaluate the constraints continuously during the matching process. It has been shown that using Persistent Analysis, the saved computational time is $n^k - r$ if the algorithm finds an unsatisfied constraint, while it evaluates the constraints on the matched input model node $r$. Furthermore, it has been shown that a transformation with Persistent Analysis never increases the computational complexity of the whole transformation process.

- I have proposed an optimized constraint validation algorithm (ORCV) that completes the RCV algorithm with the PA algorithm. I have proven that the computational complexity of the ORCV algorithm reaches the computational complexity of the RCV algorithm in the worst case only.

  Furthermore, I have shown that with ORCV the saved computational time is

  - $SC_{ORCV} \geq p \cdot \log v$ if the PA algorithm can evaluate $p$ constraints during the matching process and all of the constraints are satisfied,

  - $SC_{ORCV} \geq n^k - r$ if the PA algorithm finds an unsatisfied constraint, while evaluates the constraints on the matched input model node $r$.

Thesis II

Related publications: [109] [110] [111] [114] [116] [117] [118] [120] [122] [123] [124] [125] [127] [128] [130] [131] [140] [141].

I have supplied methods for aspect-oriented constraint management in metamodel-based model transformations, and constraint relocation and decomposition-based algorithms for constraint optimization in UML class diagram-based models.

- I have proposed a method for aspect-oriented constraint management in model transformations. This method makes possible to avoid repetitive constraints in metamodel-based model transformation rules.

- I have shown the concept of the type-based weaving and the weaving constraint-driven constraint-based weaving.
• I have introduced a new type of aspect: the constraint aspect. A constraint aspect has a structure and contains type and multiplicity conditions and weaving constraints.

• I have shown that the OCL constraints can be converted into equivalent constraint aspects. The computational complexity of the \texttt{CREATECONSTRAINTASPECT} method is $O(n)$, where $n$ is the number of the navigation steps contained by the processed constraint.

• I have introduced the concept of AND/OR clauses, and I have provided algorithms to eliminate navigation steps from the OCL constraints as a preprocessor of the real constraint validation. These steps must be executed once for the specified constraints, and their result can be reused any time to accelerate the constraint validation.

• Normalization methods are applicable not only for the constraint aspects and transformation rules but also for any UML class diagram.

• I have shown that with constraint relocation the model generated by the \texttt{NORMALIZECONSTRAINT} algorithm contains as few navigation steps as possible. It has been discussed why the constraint replacement cannot be allowed through edges that allow zero multiplicity. The computational complexity of the \texttt{NORMALIZECONSTRAINT} algorithm is $O\left(\sum \limits_{i=1}^{c} (n_i + v^3)\right)$, where $c$ denotes the number of the propagated constraints contained by the transformation rule, $n_i$ is the number of the navigation steps contained by the constraint $i$, and $v$ denotes the number of PRNs in the transformation rule.

• I have proven that applying the \texttt{DECOMPOSECONSTRAINT} algorithm, the number of the navigation steps in the constraints contained by the output model is minimal.

• I have shown that the proposed constraint relocation and constraint decomposition do not modify the result of the constraint evaluation.

• I have proposed algorithms for OCL constraint and constraint aspect propagation.

• The computational complexity of the \texttt{GLOBALCONSTRAINTWEAVER} algorithm is at most $O\left(\sum \limits_{i=1}^{c} \sum \limits_{j=1}^{s} \left(\lg v_r + n_{ij} * v^h_r + n_{ij}\right)\right)$, where $v_r$ denotes the number of the nodes in the actual transformation, $\lg v_r$ is the complexity of querying a node from the database, $c$ denotes the number of the constraints to be propagated to the transformation rules, and $s$ is the number of the transformation rules. Furthermore, $n_{ij}$ is the number of the nodes in the transformation rule $j$ with the same type as the context of the constraint $i$. $v^h_r$ is the worst case for finding an isomorphic submodel, where $k$ is the size of the submodel.

• The computational complexity of the \texttt{CONSTRAINTASPECTWEAVER} algorithm is at most $O\left(\sum \limits_{i=1}^{ca} \sum \limits_{j=1}^{s} \left(n_j^{k_i} + \sum \limits_{p=1}^{c_i} m_{jp}\right)\right)$, where $ca$ denotes the number of the constraint aspects, $s$ is the number of the transformation rules, $c_i$ is the number of the constraints contained by the constraint aspect $i$, $n_j^{k_i}$ is the complexity of the metatype-based matching (worst case), $n_j$ is the number of PRNs contained by the transformation rule $j$, and $k_i$ is the number
of PRNs contained by the constraint aspect \( i \). The \( m_{ip} \) is the number of PRNs with the metatype that corresponds to the context information of constraint \( p \).

- With the presented constraint management, not only individual transformation rules but whole transformations can be validated.

- I have proven that using constraint aspects in model transformation provides the same result as the OCL constraints.

- I have shown that working with constraint aspects is more efficient than OCL constraints. The structure of the constraint aspects facilitates to apply the metatype-driven matching to accelerate the weaving process. Furthermore, I have shown that the evaluation complexity of the propagated normalized constraint aspect reaches the evaluation complexity of the propagated OCL constraint in the worst case only.

**Thesis III**

**Related publications:** [69] [112] [115] [120] [123] [131] [132] [133] [134] [135] [136] [137] [138] [139] [141] [142].

I have introduced a visual control flow language and its termination properties. Furthermore, I have supplied a method for metamodel-based transformation rule composition that facilitates the analysis of control flow specifications.

- I have provided a new visual control flow language that supports sequencing transformation rules, branching with OCL constraints, hierarchical transformation rules, parallel execution of the rules, and iteration as well as recursion.

- I have shown that in case of rule failure the VCFL transformations do not return with error immediately. If the next element of the control flow is a decision object, then it can choose a control flow branch based on the actual input model and the value of the system variable \( \text{SystemLastRuleSucceed} \).

- I have provided algorithms to check whether a transformation contains isolated or illegal transformation rules and to validate that the OCL constraints assigned to a decision object are disjoint. The latter algorithm supports that the VCFL transformations are deterministic.

- I have suggested a method for composing metamodel-based model transformation rules. In addition, self-composing the transformation rules has been also discussed.

- Three different transformation rule structure have been introduced (tree, inserting, deleting). Let \( S \) be a transformation rule with one of the presented structure, and \( S^n \) is the transformation rule resulted by composing \( n \) times the rule \( S \). It has been proven that applying the transformation rule \( S \) \( n \) times on a finite input model is equivalent to a single execution of the rule \( S^n \).
• I have introduced rules for calculating edge multiplicities during transformation rule composition.

• I have shown that if the transformation rules $S_j, S_{j+1} \ldots S_k$ are applicable successfully for an input model, and the only composed transformation rule $S_C$ that can be created from transformation rules $S_j, S_{j+1} \ldots S_k$, then it has the same effect on the input model as the transformation rules $S_j, S_{j+1} \ldots S_k$.

• I have given conditions, and I have proven when a VCFL transformation does not terminate because of an exhaustive transformation rule or a loop.

• I have proposed an algorithm that validates the presented properties on VCFL transformations. It has been discussed that the VMTS termination algorithm cannot make decision in all cases. In certain cases the decision cannot be made based on the VCFL model only, or cannot hold for all valid input model.

• I have proven that a VCFL transformation terminates if all exhaustive transformation rule and loop terminate.

9.2 Application of the Theoretical Results

Related publications: [68] [69] [111] [114] [118] [120] [123] [125] [126] [130] [131] [136] [141] [142] [145] [146] [156].

Via the Visual Modeling and Transformation System I have shown the following.

• The model transformation validated online can be realized with constraint management and validation methods.

• The online constraint validation accelerates the whole model transformation process.

• Aspect-oriented constraint management is an efficient part of the model transformation systems. It makes the transformation rules and the constraints reusable.

• Constraint relocation and decomposition are applicable for UML class diagram-based models.

• A visual control flow language can be realized as a stereotyped UML activity diagram. Such a visual transformation language can support sequenced transformation rules, constraint-driven branching, hierarchical rules, recursion, iteration, and parallel rule execution.

• Metamodel-based model transformation rules can be composed using the presented $E$-based composing algorithm.

• Certain termination properties of the transformations can be validated offline, without input models.
9.3 Future Work

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

- Developing VMTS Aspect-UML. An extended Aspect-UML is needed to express aspect models. Pointcuts, joinpoints and advice cannot be expressed exactly using current UML tools. As aspects are context-specific and need initialize, Aspect-UML needs to be able to customize the aspect model based on different contexts. Another requirement for Aspect-UML is that the connections between aspects and classes should be detailed enough to make the tracking easy. This requires that both the advice and pointcuts are represented in the aspect model.

- Weaving aspect models into primary model. Aspects should be handled on two levels: both at the modeling level (AOM) and the programming language level (AOP). At the AOM level, aspects are identified and woven together by AOM weaving to verify and optimize them. However, models are not woven together for the purpose of code generation based on a combined model. The actual weaving is done on CodeDOM models. Fig. 9.1 introduces the block diagram of the process.

- Automatic model maintenance by validated model transformations. If a metamodel has been changed, a transformation should modify the instance models in accordance with the modified metamodel, it should contain not allowed types or attributes.

- Defining not only constraints, but (complex) attributes and object behaviors (methods) separately from model elements. They can also propagated later to the required places. They can be reused and their management also become easier and consistent.
Chapter 9. Conclusions

- Developing further optimization algorithms to support more efficient constraint evaluation.
- Developing more algorithms and considerations related to the termination and behavior of the transformations.
- Optimizing algorithm composing the transformation rule with further constraint and multiplicity examining algorithms.
- Extending the transformation capabilities of VMTS, based on the suggestions of OMG QVT [161].
- The implementation of VMTS Transformation Debugger is in progress. It makes possible to place break-points into the control flow model and to follow the whole transformation step-by-step. A debugger window shows the value of the actual transformation rule, PRN properties, and the result of the constraint evaluations. Future work includes the completion of the debugger with online parameter, attribute value, and constraint modification possibilities. This means that the modeler can modify the actual properties appearing in the debugger watch during the transformation process. Moreover, allowing the online constraint propagation (dynamic constraint weaving) and deletion as well.
- Focusing on the modeling level, rather than source code on the implementation level developing further methods to support validation and verification of model transformations [148]:
  - Automatic comparison of models. During model transformation testing, comparison between two models must be accomplished to determine if there is a inconsistency between the models. Existing graph matching algorithms are often too expensive for such a task [96]. A model transformation testing engine requires an efficient and applicable model comparison algorithm.
  - Visualization of model differences. To support visualizing the results of model comparison, efficient visual methods are needed to highlight model differences. For example, well-defined lines, symbols and colors can be used to indicate whether a model element is missing or redundant.

After determining an error that exists in a model transformation, the already mentioned debugging tool for model transformations can offer support for isolating the cause of a transformation error.

The field of model transformation-driven model-based development and aspect-oriented software development are a constantly developing area. The results of this thesis are recent results of a relatively new discipline, and hopefully, they will be a useful part of the model-driven and aspect-oriented software development.
Appendix

Detailed Proofs

Proof for Proposition 5.4

Proof. Let $H$ be an optional input model (UML class diagram), let $C$ be an OCL constraint which is propagated to $H$. Running the NormalizeConstraint algorithm, it results that the $A$ is the optimal node to which the OCL constraint $C$ should be assigned. But assume that there exists another node ($B$) for which the following holds: if one links the $C$ constraint to the node $B$ and updates the navigation paths of $C$, then the $C$ constraint contains less navigation steps than if it had been propagated to the node $A$.

The NormalizeConstraint algorithm visits all the nodes in the input model $H$, it calculates for all nodes what would be the number of the navigation steps if the constraint were relocated to the actual node and the navigation paths would be updated. Therefore, if the node $B$ was better in case of the constraint $C$, then it would be found by the NormalizeConstraint algorithm. This contradicts the assumption.

Proof for Proposition 5.6

Proof. Let $H$ be an arbitrary input model (UML class diagram), and let $C$ be an OCL constraint which is propagated to $H$. Running the DecomposeConstraint algorithm results the model $H_1$ with normalized constraints. Assume that exists a $H_2$ normalized model of $H$ which contains less navigation steps than $H_1$.

The DecomposeConstraint algorithm eliminates all the navigations except if a constraint requires property values from different nodes. If a constraint requires property values from different nodes, the DecomposeConstraint algorithm calls the NormalizeConstraint algorithm to reduce the number of the navigation steps. It is proven in Proposition 5.4 that using the NormalizeConstraint algorithm, the number of the navigation steps in the constraints contained by the output model is minimal. That contradicts the assumption.
Proof for Proposition 5.7

Proof. Let \( H \) be an arbitrary UML class diagram-based input model, let \( H_1 \) be the result model of the NormalizeConstraint algorithm, and let \( H_2 \) be the result model of the DecomposeConstraint algorithm after the constraint normalization. Assume that evaluating constraints contained by \( H_1 \) or \( H_2 \) produces different value than evaluating constraints contained by \( H \).

Both the DecomposeConstraint and the NormalizeConstraint algorithms update the navigation paths and the context information of the constraints, but they do not modify the constraint conditions (see the pseudo code and the descriptions of the algorithms). That contradicts the assumption.

\[ \square \]

Proof for Proposition 5.8

Proof. Assume two identical transformation rules \( S_1 \) and \( S_2 \), and two identical input models \( H_1 \) and \( H_2 \) as well as an OCL constraint \( C \). A constraint aspect \( CA \) is created from the OCL constraint \( C \) with the CreateConstraintAspect algorithm. Furthermore, assume that \( C \) is propagated to \( S_1 \) and \( CA \) to \( S_2 \), and apply \( S_1 \) to \( H_1 \) and \( S_2 \) to \( H_2 \). Then during the execution of \( S_1 \) and \( S_2 \) the evaluations of the propagated constraints (\( C \) and the constraint contained by \( CA \)) are different, namely, one of them returns true and the other returns false.

During the propagation of the constraint \( C \), the algorithm checks all the possible places of the transformation rule \( S_1 \) where the constraint can be assigned. The algorithm selects the PRNs with the metatype which corresponds to the context information of the constraint \( C \), it checks those places if the structure of the transformation rule \( S_1 \) satisfies the pattern required by the navigation paths of the constraint \( C \), and assigns the constraint \( C \) to the appropriate places only.

The CreateConstraintAspect algorithm identifies the root node type of the constraint aspect pattern by the context of the constraint \( C \). It creates the root node (line 2), walks through the navigation paths of the \( C \) and creates the pattern of the \( CA \) constraint aspect (lines 3-5). Furthermore, the algorithm propagates the OCL constraint \( C \) to the root node of the constraint aspect (line 6). Therefore, the pattern of \( CA \) is equal to the pattern which is required during the propagation of the constraint \( C \).

As a conclusion: (i) In the case of the constraint \( C \) the navigation paths are checked by the propagation algorithm while linking the constraints. (ii) The pattern of the \( CA \) includes the structure information as well, and the propagation of the \( CA \) constraint aspect is achieved by this structure information. This means that the same pattern is checked during the propagation processes, therefore the constraints are propagated to the same places.

This contradicts the assumption. Therefore the transformation rules with the propagated OCL constraint \( C \) and the constraint aspect \( CA \) cannot produce different result models.

\[ \square \]

Proof for Proposition 5.9

Proof. Assume two identical transformation rules \( S_1 \) and \( S_2 \), two identical input models \( H_1 \) and \( H_2 \) and a constraint aspect \( CA \). A normalized constraint aspect \( CA' \) created from the
constraint aspect $CA$ with the NormalizeConstraint algorithm. Assume that we propagate $CA$ to $S_1$ and $CA'$ to $S_2$ and apply $S_1$ to $H_1$ and $S_2$ to $H_2$. Then during the execution of $S_1$ and $S_2$, the evaluations of the propagated constraints (the constraint contained by the $CA$ and the constraint contained by the $CA'$) are different i.e. one of them returns true and the other returns false.

The NormalizeConstraint algorithm processes the OCL constraints propagated to the constraint aspect individually. The main for all loop (lines 2-17) examines the navigation paths of the actual constraint. (i) If all the paths have the same destination node, then the algorithm removes the navigations from the constraint and relocates it to the destination node (lines 5-6). (ii) Otherwise the constraint has more than one navigation path, and the destination nodes of the paths are different. In this case the algorithm checks what would happen if the constraint were relocated to another PRN (lines 7-14). The check is achieved with a calculation which takes all the PRNs contained by the constraint aspect into consideration. This calculation sums the number of the navigation steps, which is necessary in aggregate to reach all destination nodes of the original constraint ($C$) from the new place. The algorithm stores the most appropriate node (optimalNode) (line 14) and, finally, it updates the navigation paths and relocates the constraint to the optional node (lines 15-17). Updating the navigation paths means that the navigation paths of a constraint are modified such that the constraint refers to the same destination nodes from its new place as well. The relocation of a constraint means that the constraint is removed from its original place and linked to its new PRN.

The NormalizeConstraint algorithm does not change the structure of the constraint aspect, the propagation of the constraint aspect $CA$ and the normalized constraint aspect $CA'$ is accomplished based on their structure information, thus, the constraints are propagated to the same places. Furthermore, the NormalizeConstraint algorithm updates the navigation paths and relocates the constraints, but it does not modify the conditions of the constraints, which results that the same conditions are required both in the case of the constraint aspect $CA$ and the normalized constraint aspect $CA'$.

This contradicts the assumption. Therefore, the transformation rules with the propagated constraint aspect $CA$ and the normalized constraint aspect $CA'$ cannot produce different result models.

Proof for Proposition 5.10

Proof. It follows from Proposition 5.8 and 5.9 that the OCL constraint $C$ and the normalized constraint aspect $CA'$ created with the CreateConstraintAspect and NormalizeConstraint algorithms from $C$ are equivalent: The path information from the OCL expression has been built into the pattern of the constraint aspect. The OCL constraint contains the path information in its text, the constraint aspect contains the same path information in its pattern. Thus, both of them contain the same conditions for the same destination nodes. The equivalence relation is transitive, therefore this statement follows from Proposition 5.8 and 5.9.
Proof for Proposition 5.11

Proof. Assume two identical input models $H_1$ and $H_2$, transformation $T_1$ is applied to $H_1$ and $T_2$ to $H_2$. After each transformation step the success of the actual transformation rules are compared. In step $n$, $T_{1n}$ and $T_{2n}$ produce different results - one of them is successful, but the other fails because of a constraint failure.

Two cases must be separated: (i) If the rule $n$ modifies the property checked by the constraint $C$, then the constraint $C$ is propagated to the transformation $T_{1n}$ by the GCW algorithm. Hence both transformations $T_1$ and $T_2$ contain the constraint $C$ in the rule $n$, therefore, the result of the $T_{1n}$ and $T_{2n}$ rules cannot be different. (ii) If the rule $n$ does not modify the checked property (the $T_{1n}$ does not contain the constraint $C$), the constraint evaluation in $T_{2n}$ rule cannot be unsuccessful. This contradicts the assumption. 

Proof for Proposition 5.14

Proof. Assume two identical input models $H_1$ and $H_2$, transformation $T_1$ is applied to $H_1$ and $T_2$ to $H_2$. After each transformation step, the results of the actual transformation rules are compared. In step $n$, $T_{1n}$ and $T_{2n}$ produce different results - one of them is successful, but the other fails because of a constraint failure.

Based on the Proposition 5.10 the OCL constraint $C$ and the normalized constraint aspect $CA'$ are equivalent.

(i) The inputs of the GCW algorithm are OCL constraint(s) and a transformation, let them be constraint $C$ and transformation $T_1$. GCW checks all the possible places of the passed transformation rules ($T_{11}...T_{1n}$). These are places to which the constraint can be assigned. The algorithm selects the PRNs with metatype that corresponds to the context information of the constraint $C$ (line 4). It evaluates the weaving constraints (line 5) and checks the resulted places if the structure of the actual transformation rule satisfies the pattern required by the navigation paths of the constraint $C$ (line 6). For the PRNs selected by their surrounding structure, the algorithm checks if their rule requires the constraint $C$ to be propagated to any of them. The algorithm decides whether it is the first or the last rule in the transformation or whether it can modify the property contained by the constraint $C$ (IsRequiredToWeave method). Finally, the GCW algorithm propagates the constraint $C$ to the required places (line 8).

(ii) The inputs of the CAW algorithm are constraint aspect(s) and a transformation. They are denoted with the constraint aspect $CA$ and the transformation $T_2$. The CAW algorithm checks the transformation rules individually. In each rule, it searches for matches by the pattern of $CA$ (line 4), and evaluates weaving constraints on them (line 5). For each constraint ($CA.C_1...CA.C_n$) contained by $CA$, the algorithm selects the PRNs from the matches with the metatype which corresponds to the context information of the actual constraint (line 7). Similarly to the GCW algorithm, CAW also uses the IsRequiredToWeave method to decide if a transformation rule requires $CA$ to be propagated to the actual PRN (line 8). If at least one of the constraints ($CA.C_1...CA.C_n$) contained by $CA$ requires to be propagated, the whole constraint aspect is linked (line 9).
As it has been presented, the difference between the GCW and CAW algorithms is that the GCW checks the pattern of the transformation rule according to the text of the OCL constraint, while the CAW utilizes that the constraint aspect includes the pattern in its structure. This contradicts the initial assumption.

\[\square\]

**Proof for Proposition 6.32**

*Proof.* Firstly we examine the case when the transformation rule \(S_T\) applied twice for the input model \(G_0\). A tree structure rule adds a new part to the input model. It is optional to which part of LHS and thus to which part of the input model the newly created part is glued. The only requirement is not to violate the tree structure condition: not to generate circles into the input model. Fig. A.1 depicts a sample tree structure transformation rule. LHS contains two PRNs (\(A\) and \(B\)), furthermore, the new part containing \(C\) type nodes are glued to \(B\) type nodes. Note that we can choose the point optionally to which the new part is glued.
Composing a tree structure transformation rule with itself means multiplicity calculation and modification. The composition results the following: (i) the structure of the original rule $S_T$ and the composed rule $S_T^2$ are the same, (ii) in the case of the nodes affected by the external causalities, the multiplicities remain unmodified, furthermore, (iii) for nodes not affected by external causalities, the multiplicities on the linking edges are doubled ($r$ and $p..s$ are replaced with $r..2r$ and $p..2s$) (Table 6.2).

In Fig. A.1, the composition of rule $S_T$ with itself follows the arrangement and notations of Fig. 6.4. $L_1$, $K_1$, and $R_1$ represent the first execution of rule $S_T$ on the instance layer. Similarly, $L_2$, $K_2$, and $R_2$ represent the second execution of rule $S_T$ on the instance layer. External causalities are defined between PRNs $A \in S_T^{RHS}$ and $A \in S_T^{LHS}$, furthermore, between $B \in S_T^{RHS}$ and $B \in S_T^{LHS}$. Therefore, both execution of rule $S_T$ matches the same $A$ and $B$ type nodes: $L_1$ and $L_2$ contain the same instance nodes. In Fig. 6.4 $p_1 = (L_1 \xrightarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ and $p_2 = (L_2 \xrightarrow{l_2} K_2 \xrightarrow{r_2} R_2)$ are productions on instance layer. The $E$ graph is an $E$-dependency relation for $p_1$ and $p_2$ with morphisms $e_1 : R_1 \rightarrow E$ and $e_2 : L_2 \rightarrow E$. (1) and (2) are pushout complements over $K_1 \xrightarrow{r_1} R_1 \xrightarrow{e_1} E$ and $K_2 \xrightarrow{l_2} L_2 \xrightarrow{e_2} E$, furthermore, (3) and (4) are pushouts. The $E$, $C_1$, $C_2$, $L$, and $R$ graphs on the instance layer are calculated based on the Concurrency Theorem [54]. The metamodels $LM$ and $RM$ that describe $L$ and $R$ are correspond to LHS and RHS of the composed rule $S_T^2$: LHS of $S_T^2 = LM$ and RHS of $S_T^2 = RM$. External causality definitions result that there is only one possible composition of the rules that makes the transformation execution unambiguous on the instance layer. Consequently, applying the rule $S_T$ twice is equivalent to applying the rule $S_T^2$ once.

Now assume the case that a transformation rule $S_T^k$ is resulted from the $k$ times composition of the rule $S_T$, and also assume that applying the rule $S_T$ $k$ times on the input model $G_0$ is equivalent to a single application of the rule $S_T^k$. Now we examine the next step, the composition of the transformation rules $S_T^k$ and $S_T$. After finding the mapping based on the external causalities between RHS nodes of the rules $S_T^k$ and LHS nodes of the rules $S_T$, the multiplicities should be modified based on the rules presented in Table 6.2. On the instance layer, because of the external causalities, the repeated execution of the rule results that the newly created part is glued again to the same input model nodes, which raises the amount of the nodes glued to the same $B$ type nodes. Since the external causalities provide the unambiguous rule execution on the instance layer, and we consider the multiplicity calculation rules of the rule composition, executing the rule $S_T$ $k + 1$ times on the input model $G_0$ is equivalent to a single application of the rule $S_T^{k+1}$. Consequently, the $n$ times application of rule $S_T$ is equivalent to the simple execution of rule $S_T^n$. \hfill \Box

**Proof for Proposition 6.34**

*Proof.* Firstly we examine the case when the transformation rule $S_I$ applied twice for the input model $G_0$. Nodes added by an inserting structure rule to an input model are inserted between two already existing nodes. It is optional which LHS PRNs and thus which input model nodes are selected to insert the newly created nodes between them. Fig. A.2 depicts a sample inserting structure transformation rule. LHS contains two PRNs ($A$ and $B$), furthermore, the new $C$ type nodes are inserted between PRNs $A$ and $B$. In the current case there are only two PRNs,
therefore, we should choose them to insert the new nodes between them, but in general the nodes can be chosen optionally.

Composing a tree structure transformation rule with itself means multiplicity calculation and modification. The structure of the original rule $S_I$ and the composed rule $S_I^2$ are the same, and the multiplicities are created based on Table 6.2.

In Fig. A.2, the composition of rule $S_I$ with itself follows the arrangement and notations of Fig. 6.4. $L_1$, $K_1$, and $R_1$ represent the first execution of rule $S_I$ on the instance layer. Similarly, $L_2$, $K_2$, and $R_2$ represent the second execution of rule $S_I$ on the instance layer. External causalities are defined between PRNs $A \in S_I^{RHS}$ and $A \in S_I^{LHS}$, furthermore, between $B \in S_I^{RHS}$ and $B \in S_I^{LHS}$. Therefore, both execution of rule $S_I$ matches the same $A$ and $B$ type nodes: $L_1$ and $L_2$ contain the same instance nodes. The $E$, $C_1$, $C_2$, $L$, and $R$ graphs on the instance layer are calculated based on the Concurrency Theorem [54]. The metamodels $LM$ and $RM$ that describe $L$ and $R$ are correspond to LHS and RHS of the composed rule $S_I^2$: LHS of $S_I^2 = LM$ and RHS of $S_I^2 = RM$. External causality definitions result that there is only one
possible composition of the rules that makes the transformation execution unambiguous on the instance layer. Consequently, applying the rule $S_I$ twice is equivalent to applying the rule $S_I^2$ once.

Now assume the case that a transformation rule $S_I^k$ is resulted from the $k$ times composition of the rule $S_I$, and also assume that applying the rule $S_I$ $k$ times on the input model $G_0$ is equivalent to a single application of the rule $S_I^k$. Now we examine the next step, the composition of the transformation rules $S_I^k$ and $S_I$. External causalities define the mapping between RHS nodes of the rules $S_I^k$ and LHS nodes of the rules $S_I$, furthermore, the multiplicities should be modified based on the rules presented in Table 6.2. On the instance layer, because of the external causalities, the repeated execution of the rule results that the newly created nodes are repeatedly inserted into the input model between the same $A$ and $B$ type nodes. This modifies the cardinality of the newly inserted nodes, and therefore, the number of the neighbors of the matched nodes. Since the external causalities provide the unambiguous rule execution on the instance layer, and we consider the multiplicity calculation rules of the rule composition, executing the rule $S_I$ $k + 1$ times on the input model $G_0$ is equivalent to a single application of the rule $S_I^{k+1}$. Consequently, the $n$ times application of rule $S_I$ is equivalent to the simple execution of rule $S_I^n$.

**Proof for Proposition 6.36**

*Proof.* Firstly we examine the case when the transformation rule $S_D$ applied twice for the input model $G_0$. A deleting structure rule removes at least one input model node. It is optional which LHS PRNs and thus which input model nodes are selected for deletion. Fig. A.3 depicts a sample deleting structure transformation rule. LHS contains two PRNs ($A$ and $B$), furthermore, the rule describes that $B$ type nodes with the related edges are removed. Note that we can choose the nodes optionally that should be deleted.

Composing a deleting structure transformation rule with itself means multiplicity calculation and modification. The structure of the original rule $S_D$ and the composed rule $S_D^2$ are the same, and the multiplicities are created based on Table 6.2.

In Fig. A.3, the composition of rule $S_D$ with itself follows the arrangement and notations of Fig. 6.4. $L_1$, $K_1$, and $R_1$ represent the first execution of rule $S_D$ on the instance layer. Similarly, $L_2$, $K_2$, and $R_2$ represent the second execution of rule $S_D$ on the instance layer. There is one external causality defined between PRNs $A \in S_D^{RHS}$ and $A \in S_D^{LHS}$. Therefore, both execution of rule $S_D$ matches the same $A$ type nodes: $L_1$ and $L_2$ contain the same $A$ type instance nodes. The $E$, $C_1$, $C_2$, $L$, and $R$ graphs on the instance layer are calculated based on the Concurrency Theorem [54]. The metamodels $LM$ and $RM$ that describe $L$ and $R$ are correspond to LHS and RHS of the composed rule $S_D^2$: LHS of $S_D^2 = LM$ and RHS of $S_D^2 = RM$. External causality definition results that there is only one possible composition of the rules that makes the transformation execution unambiguous on the instance layer. Consequently, applying the rule $S_D$ twice is equivalent to applying the rule $S_D^2$ once.

Now assume the case that a transformation rule $S_D^k$ is resulted from the $k$ times composition of the rule $S_D$, and also assume that applying the rule $S_D$ $k$ times on the input model $G_0$ is equivalent to a single application of the rule $S_D^k$. Now we examine the next step, the composition
of the transformation rules $S_D^k$ and $S_D$. External causality defines the mapping between RHS nodes of the rules $S_D^k$ and LHS nodes of the rules $S_D$, furthermore, the multiplicities should be modified based on the rules presented in Table 6.2. On the instance layer, because of the external causality, the repeated execution of the rule results that the deleted nodes are the neighbors of the same $A$ type node. This modifies the number of the $B$ type neighbors of the actually matched $A$ type node. Since the external causality provide the unambiguous rule execution on the instance layer, and we consider the multiplicity calculation rules of the rule composition, executing the rule $S_D k + 1$ times on the input model $G_0$ is equivalent to a single application of the rule $S_D^k$. Consequently, the $n$ times application of rule $S_D$ is equivalent to the simple execution of rule $S_D^n$.

\begin{proof}
There are no external causalities defined, therefore, each application of rule $S$ can match either the same or different part of the input model as the previous one. Thus the rules executed one after another can work on the different part of the input model, therefore the transformation on the instance layer become ambiguous. Examples can be found in Figures 6.5b and 6.7b.
\end{proof}
Proof for Proposition 6.38

*Proof.* In VMTS, the match found for LHS in the finite input model \((G_1)\) is maximal, no more links can be added to the match, which is also a match for the actual LHS. Moreover, the edge multiplicities of the EBMR are calculated based on the Table 6.2.

The transformation is defined on the metamodel layer using metamodel-based model transformation rules, but it is performed on the instance layer. Composing metamodel-based model transformation rules means transformation rule structure and multiplicity calculation. The composition results the following. (i) The structure of the composed rule \(S_C\) is created based on the external causalities defined between transformation rules and the metatype information of the PRNs. (ii) The multiplicities are calculated based on the Table 6.2.

Metamodel-based model transformation rule composition follows the principle depicted in Fig. 6.4. \(p_1 = (L_1 \xrightarrow{l_1} K_1 \xrightarrow{r_1} R_1)\) and \(p_2 = (L_2 \xrightarrow{l_2} K_2 \xrightarrow{r_2} R_2)\) are productions on instance layer. The \(E\) graph is an \(E\)-dependency relation for \(p_1\) and \(p_2\) with morphisms \(e_1 : R_1 \rightarrow E\) and \(e_2 : L_2 \rightarrow E\). (1) and (2) are pushout complements over \(K_1 \xrightarrow{r_1} R_1 \xleftarrow{e_1} E\) and \(K_2 \xrightarrow{l_2} L_2 \xleftarrow{e_2} E\), furthermore, (3) and (4) are pushouts. The \(E\) graph gives the mapping of the composition and selects the exact place of the transformation on the instance layer. The \(E, C_1, C_2, L,\) and \(R\) graphs on the instance layer are created based on the Concurrency Theorem [54]. Examples for the tree, inserting, and deleting structure transformation rule compositions can be found in Figures A.1, A.2, and A.3. Propositions 6.32, 6.34, and 6.36 state and prove the conditions for unambiguously composed transformation rule execution on the instance layer. Based on the unambiguous node mapping mechanism provided by external causalities, the multiplicity calculation rules, and the proofs 6.32, 6.34, and 6.36: the metamodels \(LM\) and \(RM\) that describe \(L\) and \(R\) are correspond to LHS and RHS of the composed rule \(S_C\) created with VTSComposing algorithm: \(S^L_{LHS} = LM\) and \(S^R_{RHS} = RM\). \(\Box\)
Computational Complexity
Considerations on Constraint Evaluation and Navigation

The complexity of the constraint evaluation is $O(1)$ [109]. During the navigation the cost of each navigation step is $O(\lg ev)$ if the navigation path contains only multiplicities 0..1 or 1. It is a query on the table which contains the model edges to obtain the node IDs of the appropriate adjacent nodes ($O(\lg e)$) and the adjacent nodes by their IDs have to be selected ($O(\lg v)$ for each node): $O(\lg e + m \cdot \lg v) = O(\lg e + \lg v^m) = O(\lg ev^m)$, where $m$ is the number of the selected adjacent nodes. If $m$ is 1, then it is $O(\lg ev)$.

If the navigation path contains multiplicities like 0..*, the result of a query can be a collection of nodes (maximum $v$ nodes) instead of a simple node, thus, the next step must work on the result of the previous step. Traversing the navigation paths, the metatype of the nodes needs to be taken into consideration, because it reduces the complexity of the whole evaluation process.

An OCL constraint $C$ contains any number of navigation paths. A navigation path $P$ is depicted in Fig. B.1a, $P$ contains $r$ navigation steps and traverses $r + 1$ nodes ($v_0, v_1, ..., v_r$).

Fig. B.1. (a) A navigation path, (b) Example metamodel with constraints, (c) Example model
Chapter B. Computational Complexity Considerations on Constraint Evaluation and Navigation

Let \( \text{Metatype}(v_i) \) denote the collection of the model nodes in the input model with metatype \( v_i \), where \( 0 \leq i \leq n \), and \( n \) is the number of the nodes in the input model. Let \( R_i \) be a node collection resulted from the navigation step \( i \). Since \( R_0 \) is the start node collection of the constraint \( C \), this means that the number of the nodes in collection \( R_0 \) is 1 (\( \#R_0 = 1 \), \( R_0 = \{v_0\} \)). Using the \text{SelectNodes} method, the result node collection of the navigation step \( i \) can be obtained, based on the result of the navigation step \( i - 1 \) and the nodes with metatype of the path node \( i \): \( R_i = \text{SelectNodes}(R_{i-1}, \text{Metatype}(v_i)) \). The maximum number of the nodes in the collection \( \#R_i \) is \( \#R_i = \#R_{i-1} \times \#\text{Metatype}(v_i) \). In summary, the cost of the whole navigation is maximum

\[
\text{Cost}_R = \log e \times \sum_{i=1}^r \left( \#R_{i-1} \times (\log e + \#\text{Metatype}(v_i) \times \log v) \right).
\] (B.1)

The first part of the formula (\( \log e \times \)) is the cost of the step 0 (to select the start node), and the second part is the cost of the steps from 1 up to \( r \). There are \( r \) navigation steps, in the navigation step \( i \) there are \( \#R_{i-1} \) queries on the table of edges and \( \#R_i = \#R_{i-1} \times \#\text{Metatype}(v_i) \) selects on the table of nodes.

Formula B.1 is only an approximation, it is the maximum value of the real cost of the navigation. This approach utilizes the different metatypes of the nodes, therefore the worst case is if all the model nodes have the same metatype.

To approximate the real cost of the navigation better \( \text{MetatypeNeighbour}(v_{ij}) \) operation can be defined which retrieves not all of the model nodes with the metatype \( v_i \), but only the adjacent nodes of the collection node \( j \) with metatype \( v_i \). Based on it and on \( \#R_i' = \sum_{j=1}^{\#R_{i-1}} \#\text{MetatypeNeighbour}(v_{ij}) \), the modified formula is

\[
\text{Cost}_{R'} = \log e \times \sum_{i=1}^r (\#R_{i-1}' \times \log e) + \sum_{j=1}^{\#R_i'} (\#\text{MetatypeNeighbour}(v_{ij}) \times \log v) = \quad \text{(B.2)}
\]

\[
\log e \times \sum_{i=1}^r (\#R_{i-1}' \times \log e + \#R_i' \times \log v)
\] (B.3)

Example calculations

In Fig. B.1b, an example metamodel with constraints is depicted, and in Fig. B.1c an example instance model is illustrated. In this section, the cost of the process to traverse the navigation paths is calculated in three different ways: manually, which shows the minimal necessary cost, and with the Formula B.1 and B.3. In Fig. B.1b, the dashed lines denote that constraint \( \text{Const1} \) navigates from node \( A \) through node \( B \) to node \( C \), and constraint \( \text{Const2} \) navigates from node \( A \) through nodes \( B \) and \( C \) to node \( D \).
Chapter B. Computational Complexity Considerations on Constraint Evaluation and Navigation

Example 1 - Const1

- Manual calculation.

<table>
<thead>
<tr>
<th>Step 0</th>
<th>Cost</th>
<th># Result Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 0</td>
<td>$\lg e v$</td>
<td>1</td>
</tr>
<tr>
<td>Step 1</td>
<td>$\lg e v^2$</td>
<td>2</td>
</tr>
<tr>
<td>Step 2</td>
<td>$\lg e^2 v^4$</td>
<td>4</td>
</tr>
<tr>
<td>$\sum$</td>
<td>$\lg e^4 v^7$</td>
<td></td>
</tr>
</tbody>
</table>

Table B.1. The calculation table of the navigations contained by Const1 (Fig. B.1b and Fig. B.1c)

In step 0, the start node (A1) is selected ($\lg e v$). In step 1, the adjacent nodes of the start node with metatype $B$ are queried, the cost of it is $\lg e + 2 \cdot \lg v = \lg e v^2$. The result of step 1 is a collection with two nodes ($R_1 = \{B1, B2\}$). Finally, in step 2 the neighbors of the $B1$ and $B2$ nodes are selected with metatype $C$ ($R_1 = \{C1, C2, C3, C4\}$). The cost of step 2 is $2 \cdot \lg e + 4 \cdot \lg v = \lg e^2 v^4$, therefore the whole cost is $\lg e^4 v^7$, which means that there are four selects on the table of edges and seven queries on the table of nodes.

- Calculation using Formula B.1.
  
  $\#R_0 = 1, \#R_1 = 2, \#R_2 = 8, \#\text{METATYPE}(v_0) = 1, \#\text{METATYPE}(v_1) = 2, \#\text{METATYPE}(v_2) = 4,$
  
  $$\text{Cost}_R \leq \lg e v + \sum_{i=1}^{2} (\#R_{i-1} \cdot (\lg e + \#\text{METATYPE}(v_i) \cdot \lg v)) =$$
  
  $$\lg e v + (\lg e + 2 \cdot \lg v) + (2 \cdot (\lg e + 4 \cdot \lg v)) = \lg e^4 v^{11}$$

- Calculation using Formula B.3.
  
  $\#R'_0 = 1, \#R'_1 = 2, \#R'_2 = \#R'_{21} + \#R'_{22} = 2 + 2 = 4,$
  
  $$\text{Cost}_{R'} = \lg e v + \sum_{i=1}^{2} (\#R'_{i-1} \cdot \lg e + \#R'_i \cdot \lg v) =$$
  
  $$\lg e v + (\lg e + 2 \cdot \lg v) + (2 \cdot \lg e + 4 \cdot \lg v) = \lg e^4 v^7$$

Example 2 - Const2

- Manual calculation.

<table>
<thead>
<tr>
<th>Step 0</th>
<th>Cost</th>
<th># Result Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 0</td>
<td>$\lg e v$</td>
<td>1</td>
</tr>
<tr>
<td>Step 1</td>
<td>$\lg e v^2$</td>
<td>2</td>
</tr>
<tr>
<td>Step 2</td>
<td>$\lg e^2 v^4$</td>
<td>4</td>
</tr>
<tr>
<td>Step 3</td>
<td>$\lg e^4 v^9$</td>
<td>3</td>
</tr>
<tr>
<td>$\sum$</td>
<td>$\lg e^6 v^{19}$</td>
<td></td>
</tr>
</tbody>
</table>

Table B.2. The calculation table of the navigations contained by Const2 (Fig. B.1b and Fig. B.1c)
• Calculation using Formula B.1.

\[ \#R_3 = 56, \ #\text{METATYPE}(v_3) = 7, \]

\[
\text{Cost}_R \leq \lg ev + \sum_{i=1}^{3} (#R_{i-1} \ast (\lg e + \#\text{METATYPE}(v_i) \ast \lg v)) = \\
\lg ev + (\lg e + 2 \ast \lg v) + (2 \ast (\lg e + 4 \ast \lg v)) + (8 \ast (\lg e + 7 \ast \lg v)) = \lg e^{12}v^{67}
\]

• Calculation using Formula B.3.

\[
\#R'_3 = \#R'_{31} + \#R'_{32} + \#R'_{33} + \#R'_{34} = 1 + 0 + 2 + 0 = 3, \\
\text{Cost}_{R'} = \lg ev + \sum_{i=1}^{3} (#R'_{i-1} \ast \lg e + #R'_{i} \ast \lg v) = \\
\lg ev + (\lg e + 2 \ast \lg v) + (2 \ast (\lg e + 4 \ast \lg v)) + (4 \ast (\lg e + 3 \ast \lg v)) = \lg e^{8}v^{10}
\]
<table>
<thead>
<tr>
<th>Tool</th>
<th>Control Flow</th>
<th>Constraints in the rule</th>
<th>Attribute transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMTS</td>
<td>Stereotyped activity diagrams</td>
<td>Instantiation + OCL</td>
<td>XSLT</td>
</tr>
<tr>
<td>GReAT</td>
<td>Deterministic, non-deterministic, recursion</td>
<td>OCL</td>
<td>C-based attribute mapping language</td>
</tr>
<tr>
<td>AGG</td>
<td>Layers (exhaustive or once, loop)</td>
<td>JAVA, NAC</td>
<td>JAVA</td>
</tr>
<tr>
<td>PROGRES</td>
<td>Imperative and declarative, transactions</td>
<td>Attribute constraints, cardinality, negative edge</td>
<td>Built-in or host programming language (esp. C)</td>
</tr>
<tr>
<td>VIATRA</td>
<td>ASM</td>
<td>Graph pattern</td>
<td>ASM statements (built in support for basic JAVA types)</td>
</tr>
<tr>
<td>ATOM3</td>
<td>Layers with priorities, sequencing by priority. Parallel execution of non-overlapping matches.</td>
<td>Generalized NAC, application conditions.</td>
<td>Python</td>
</tr>
<tr>
<td>FUJABA</td>
<td>Story diagrams</td>
<td>Story diagrams, JAVA</td>
<td>Story diagrams</td>
</tr>
</tbody>
</table>

Table C.1. Comparison table of control flow, constraint and attribute transformation support for model transformation tools
Verification Tools

Model checking is increasingly popular for software verification. Model checking is one of the few verification techniques that have shown their benefits in practice. The successes are mainly limited to hardware verification, because it has been recognized that software has some features that make the problem harder. Primary among those features is the dynamic nature of software (e.g. dynamic memory allocation).

The theoretical basics of verifying graph transformation systems by model checking have been presented in [84]. It has been shown that graphs can be interpreted as states and rule applications as transitions in a transition system.

A theoretical framework is given in [17] with aims at analyzing a special class of hypergraph rewriting systems by a static analysis technique based on approximative foldings and unfoldings of a special class of Petri nets. This work has been extended in [16] to provide a precise (McMillan-style) unfolding strategy.

In [47] an object-based graph grammar is used for modeling object-oriented systems and a translation into SPIN [86] to carry out model checking is defined. A restricted structure for graph transformation rules is allowed that is tailored to model message calls in object-oriented systems. The framework relies on high-level SPIN/Promela constructs.

The related part of this section gives background information on four verification tools and approaches.

D.1 GROOVE

The GROOVE approach [168] [169] [170] uses the core concepts of graphs and graph transformations all the way through during model checking. This means that states are explicitly represented and stored as graphs, and transitions as applications of graph transformation rules; moreover, properties to be checked should be specified in a graph-based logic, and graph-specific model checking algorithms should be applied.

This approach implies that very little of the theory and tool development for traditional model checkers can be applied immediately, since the most basic concept, namely the underlying model, has been extended drastically.
The benefits of the GROOVE approach are the following: (i) There is no a priori upper bound to the size of the graphs. (ii) There is an implicit symmetry check through the identification of isomorphic graphs. (iii) No pre- or postprocessing is necessary to apply the GROOVE tool to a given graph transformation system, or to translate the results of the model checking back into graphs; (iv) Existing graph transformation theory can be directly brought to bear upon the tool.

The essential disadvantage of this approach is that the huge part of existing research in traditional model checking is only indirectly applicable.

D.2 CheckVML

CheckVML approach [170] [173] [195] [196] exploits off-the-shelf model checker tools like SPIN for the verification of graph transformation systems. It translates a graph transformation system parameterized with a type graph and an initial graph into its Promela equivalent to carry out the formal analysis in SPIN. Furthermore, property graphs are also translated into their temporal logic equivalents.

The benefits of the CheckVML approach are the following: (i) It considers typed and attributed graphs which fits well to the metamodeling philosophy of UML and other modeling languages. (ii) The size of the state vector depends only on the dynamic model elements while immutable static parts of a model are not stored in the state vector. This is a typical case for dataflow like systems. (iii) It can be easily adapted to various back-end model checker tools.

The essential disadvantage of the approach is that dynamic model elements easily blow up both the verification model and state space. Moreover, symmetries in graphs can be handled for only very limited cases.

A comparison of the CheckVML and GROOVE can be found in [170].

D.3 Augur

Augur [100] [101] is a tool for the verification of systems described by graph transformations using approximated unfoldings. The obtained over-approximation consists of an underlying hypergraph and a Petri net. Properties of graph transformation systems can be verified by analyzing the approximation, using regular expressions and coverability checking for Petri nets.

Augur takes as input language a simple yet expressive specification language: graph transformation systems (GTS). Augur approximate GTSs by Petri nets, which are a conceptually simpler formalism and for which several verification techniques have already been developed. More specifically, the tool is based on an approximate unfolding technique for GTSs.

Augur facilitates to: (i) Compute approximated unfoldings of graph grammars specified in GTXL (Graph Transformation Exchange Language), a XML standard for graph transformation systems. The resulting Petri graph is then written to a file in the GXL (Graph Exchange Language) format. (ii) It is also possible to compute subsequently better approximations disallowing folding steps until after depth \( k \). (iii) The input graph transformation and the output can be visualized using the GraphViz package [76].
D.4 OBGG

Object-Based Graph Grammars (OBGG) [46] is a formal specification language suitable for modeling concurrent object-based systems.

In [49], a visual formal specification language suitable for specifying concurrent object-based systems was defined. The language itself is a restriction of graph grammars, called Object-Based Graph Grammars (OBGG). Models in OBGG can be analyzed through verification [48] and simulation [45]. Moreover, starting from an OBGG model code can be generated for execution in a real environment, following a straightforward mapping to Java code.

OBGG is based on the message passing mechanism for communication, and follows the asynchronous computation model, and the language is also suitable for the specification of distributed systems. Thus, in order to deal with common aspects found in distributed environments, like fault-tolerance, an approach is given to consider classical failure descriptions for distributed systems (e.g. crash failure), allowing to reason about a given OBGG model in the presence of a selected failure.
Bibliography


154


