Model-Driven Development of Model Transformations

Citation for published version (APA):

van Gorp, P. M. E. (2008). *Model-Driven Development of Model Transformations*. [Phd Thesis 4 Research NOT TU/e / Graduation NOT TU/e], University of Antwerp]. Universiteit van Antwerpen.

Document status and date:

Published: 01/01/2008

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Model-Driven Development of Model Transformations

Pieter Van Gorp

Promotor: Prof. dr. Dirk Janssens
Co-promotor: Prof. dr. Serge Demeyer

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Wetenschappen

Acknowledgements

The priceless support of many has resulted in the successful completion of this thesis. I am now able to hold this document and enjoy nice memories of five years of doctoral research.

“Regrets, I’ve had a few; But then again, too few to mention.” [6]

Therefore, it is a pleasure to thank the many people who supported this challenging project.

I would like to thank my parents for giving me the motivation and resources to study at the university. Thanks to all my family members for always being there when I needed them the most. Thanks to Nena for supporting me in countless ways, in particular when I was completing this text.

Thanks to Dirk Janssens and Serge Demeyer for supervising my doctoral research. You have given me the freedom to pursue my personal research interests, you have taught me how to write scientific papers, you have enabled me to attend international conferences, you have recruited new team members in my research domain, and you have enabled me to align my teaching and research activities. Dirk, thanks for your insights related to graph transformation and thanks for introducing me to collaborative experts in the field. Serge, thanks for assisting me in finding a balance between theoretical and practical contributions.

Special thanks to all office and research group comrades, in particular to *Dubois*, *Skiptron* and *Method Man*. You have made my office time enjoyable for reasons that go beyond my research passion. Thanks to Hans Schippers and Olaf Muliawan for building the MoTMoT infrastructure. Hans, thanks for “playing” devil’s advocate so many times.

I am very grateful to the hosts of my research visits. In particular, thanks to Reiko Heckel, Lothar Wendehals and Robert Wagner for the warm welcome in Paderborn. Thanks to Jean Bézivin and Frédéric Jouault for the great time in Nantes. Thanks to the organizers of the Dagstuhl seminars that I enjoyed attending.

Thanks to Frank Altheide for visiting our lab and joining me on an exploration of the relations between new MDA technologies and more mature work in the domain of consistency maintenance. Frank, you have raised my awareness of things that I had been taking for granted while working in a rather closed community.

I owe my interest in enterprise computing to Jan Broeckhove and Gunther Stuer. Thanks for pointing me in this challenging direction. Unfortunately, it is impossible to thank everybody who may have influenced me throughout inspiring discussions. However, I would like to express a special gesture of gratitude to Hans Stenten, Tom Mens, Andy Schürr, Albert Zündorf, Hans Vangheluwe, Krzysztof Czarnecki, Kim Mens, Roel Wuyts, Jean-Marie Favre, Jos Warmer, Colin Atkinson, Ralf Lämmel, James Skene, Alexey Cherkhago, Simon Becker and Jordi Cabot.

Abstract

This thesis presents how evolving software models expressed in different formalisms can be kept consistent by means of an object-oriented integration of metamodeling, design by contract and graph transformation. It incrementally builds upon existing modeling languages to illustrate that the adoption of the new model-driven engineering paradigm does not force organizations to abandon their investments in more mature technologies. More specifically, this thesis shows how the new transformation languages introduced by the Model Driven Architecture standardization effort relate to fragments of the UML that have a well-known, object-oriented semantics.

The proposed techniques have emerged from several case studies that involve a wide variety of transformation challenges. Therefore, the contributions are applicable for transformation modeling *in general*: at first, it is illustrated how model evolution can be supported; secondly, the technique is applied to transform conceptual models into more computer oriented ones; finally, the technique is applied and extended in the context of model synchronization.

Contents

1	Introduction	13
1.1	Context	13
1.1.1	Model-Driven Engineering	13
1.1.2	Standardization	14
1.1.3	Summary of Contributions	15
1.2	Model Driven Architecture	16
1.2.1	The UML as a Framework for Domain Specific Visual Languages	17
1.2.1.1	UML Profile Application	17
1.2.1.2	Evaluation	19
1.2.1.3	UML Profile Definition	19
1.2.1.4	Language Support for Profiles	20
1.2.2	Meta Object Facility (MOF)	22
1.2.3	MOF Queries, Views and Transformations RFP	22
1.2.3.1	Requirements from the Request For Proposals	22
1.2.3.2	Evaluation of Existing Solution: Java Metadata Interface	23
1.3	Towards Standards for Modeling Transformations	24
1.3.1	Jézéquel's OCL Actions	24
1.3.2	Cariou's OCL Transformation Contracts	25
1.3.3	Akehurst's OCL Relations	26
1.3.3.1	Standardization of a Mapping Operator	27
1.3.3.2	Pattern for Modeling Mathematical Relations in the UML	28
1.3.4	Controlled Graph Transformation, Story Diagrams	32
1.3.4.1	Graph Transformation Basics: Left- and Right-hand Sides	33
1.3.4.2	Concrete Syntax: Merging LHS and RHS	34
1.3.4.3	Control Structures	36
1.3.4.4	Evaluation of the Story Driven Modeling Methodology	37
1.3.5	MOF Queries, Views and Transformations Standard	38
1.4	Summary and Outlook	39
2	A Taxonomy of Model Transformation	41
2.1	The Taxonomy as an organic Community Artifact	42
2.2	Classifying Transformation Types	42
2.2.1	Transformation Types	43
2.2.2	Taxonomy Elements	44
2.2.2.1	Translation or Rephrasing	44

2.2.2.2	Horizontal or Vertical	44
2.2.2.3	Syntactical or Semantical	45
2.2.3	Classification	45
2.3	Classifying Transformation Designs	46
2.3.1	Number of (Sub-)Models and Metamodels	46
2.3.1.1	First-class or Sub-model Transformation	46
2.3.1.2	Input-destructive, Input-polluting or Input-preserving	48
2.3.1.3	In-place or Out-place	49
2.3.1.4	Endogenous or Exogenous	49
2.3.1.5	Relation to Transformation Types	50
2.3.2	Technical Space	50
2.3.3	Intra- or Inter-Space	51
2.3.4	Versatility: Enforceable, Checkable, Ignorable, Prioritizable	52
2.4	Classifying Transformation Languages	52
2.4.1	Declarative or Operational	52
2.4.1.1	Change Propagation: Implicit, Explicit or Both	53
2.4.1.2	Execution Direction: Implicit, Explicit or Both	53
2.4.1.3	Rule Scheduling: Implicit, Explicit or Both	55
2.4.1.4	Restrictive or Constructive	56
2.4.2	Concrete Syntax	57
2.4.3	Decomposition Mechanisms	58
2.4.4	Genericity	58
2.4.4.1	Specialization	58
2.4.4.2	Overloading	58
2.4.4.3	Reflection	59
2.4.4.4	Higher Order Transformations	59
2.4.5	Traceability	59
2.5	Classifying Transformation Tools	60
2.5.1	CRUD: Creating/Reading/Updating/Deleting transformations	60
2.5.2	Useful and Usable	60
2.5.3	Run-Time Performance versus Developer Performance	61
2.5.4	Input/Output: In-Memory versus Serialized	61
2.5.5	Standards	61
2.6	Summary and Outlook	62

I Refactoring

65

3 Towards Automating Source-Consistent UML Refactoring

67

3.1	Refactoring: Context, Tools, Relation to the UML	68
3.2	Describing UML Refactorings as Refactoring Contracts	68
3.2.1	Refactoring Contracts	68
3.2.2	Extract Method	69
3.2.2.1	OCL Pre- and Postconditions	69
3.2.2.2	OCL Code Smell: <i>Duplicate Code</i>	71
3.2.3	Pull Up Method	72
3.2.4	Discussion	72

3.3	Applications of Automated UML Refactoring	73
3.3.1	Compose primitive refactorings	73
3.3.2	Integrate with code smell detectors	75
3.4	Related and Ongoing Work	76
3.5	Summary and Outlook	77
4	Expressing Refactorings as Story Diagrams	79
4.1	Case Study: Pull Up Method	79
4.2	Transformation Modeling using Story Diagrams	80
4.2.1	Model Transformation	81
4.2.2	Precondition	81
4.3	Cognitive Evaluation of Transformation Model	84
4.4	Deploying and Using the Transformation	87
4.5	Summary and Outlook	87
5	Standardizing Story Diagrams using UML and MOF	89
5.1	Need for Exchangeability of Story Diagrams	90
5.1.1	Problem: Implicit and Proprietary Metamodel	90
5.1.2	Solution: UML Profile for Story Diagrams	91
5.1.3	Discussion	92
5.2	Need for executability on Standard Repositories	93
5.2.1	Problem/Limitation: Fujaba Association Framework	94
5.2.2	Solution: Translating Story Diagrams to JMI Code	94
5.2.2.1	Loading Transformation Models	95
5.2.2.2	Querying Transformation Elements	95
5.2.2.3	Generation of Transformation Code	97
5.3	Example: Pull Up Method	98
5.4	Related and Future Work	99
5.4.1	Fujaba Codegen2	100
5.4.2	OCL Support	101
5.5	Summary and Outlook	101
II	Synthesis	103
6	UML2CSP:	
	a Platform Independent Transformation Model	107
6.1	Context: Model Transformation Tool Contest	107
6.2	UML-to-CSP: General and Specific Requirements	108
6.2.1	General Requirements of the Case Study	108
6.2.2	Optional Requirements of the Case Study	108
6.2.3	Additional Challenges concerning Industrial Relevance	109
6.2.3.1	Platform Independent Transformation Modeling	109
6.2.3.2	Robustness of the Mapping	111
6.3	MoTMoT: Transformation Model for UML-to-CSP	112
6.3.1	Architecture	112
6.3.2	Structure: Related Elements and Mapping Responsibilities	113

6.3.3	Behavior of the Transformation Methods	115
6.3.3.1	Transform Action Node	115
6.3.3.2	Transform Initial Node	116
6.3.3.3	Transform Final Node	116
6.3.3.4	Transform Decision Node	117
6.3.3.5	Transform Merge Node	121
6.3.3.6	Transform Fork Node	122
6.3.3.7	Transform Join Node	124
6.3.3.8	Transform IN to OUT	125
6.3.4	In-Place Transformation removing non-standard constructs	127
6.3.5	Declarative Rule Scheduling	129
6.3.5.1	Java Overloading: Implicitly Check <i>Static Types</i> of Arguments	130
6.3.5.2	Solution 1: <i>Explicitly Check Types</i> and <i>Cast Arguments</i>	130
6.3.5.3	Solution 2: Reflective Interpreter	131
6.3.5.4	Solution 3: Move the “transform” method	131
6.4	Lessons Learned	132
6.5	Summary and Outlook	133
7	Copying Subgraphs within Model Repositories	135
7.1	Motivating Example Models	136
7.2	The CM2RM Transformation	138
7.2.1	Structural model of the transformation	138
7.2.2	Behavioral model of the transformation	139
7.2.2.1	Finding a Match	140
7.2.2.2	Copying the Subgraph	141
7.2.2.3	Using the Copy	141
7.3	Subgraph Copy operator	142
7.3.1	What	142
7.3.2	How	144
7.4	Related Work	145
7.5	CM2RM Revisited	146
7.5.1	Extended Requirements for CM2RM	146
7.5.2	Extended Transformation Model	149
7.5.3	Lessons Learned	151
7.5.3.1	Traceability Metamodel	151
7.5.3.2	Copying links between Copied Elements	151
7.6	Summary and Outlook	153
8	Copy2GT: A Higher-Order Transformation Model	155
8.1	Introduction	155
8.2	Copy2GT: <i>Desugaring</i> the profile for Story Diagrams	156
8.2.1	Example Input/Output Models	156
8.2.1.1	Story Diagram	156
8.2.1.2	Story Patterns	159
8.2.2	Mapping Rules	162
8.2.2.1	Desugaring «copy»	162

8.2.2.2	Desugaring Composition Path	163
8.2.2.3	Desugaring «onCopy» on Attribute Assignments	163
8.2.2.4	Desugaring «onCopy» on «create»/«destroy» Links	163
8.3	Higher Order Transformation in Story Diagrams	164
8.3.1	Story Diagrams	164
8.3.2	Story Patterns	164
8.4	Lessons Learned	166
8.5	Summary and Outlook	167

III Model Synchronization 169

9 CAViT: Consistency Maintenance with Transformation Contracts 173

9.1	Model Synchronization using Design By Contract	174
9.1.1	Design by Contract	174
9.1.2	Contract Aware Model Transformation	174
9.1.2.1	Consistency Maintenance	175
9.1.2.2	Inconsistency Management	176
9.2	Example Transformation Models	176
9.3	Architecture of the CAViT Framework	180
9.4	Related Work	180
9.4.1	Object-Oriented Metamodeling and Consistency Maintenance	181
9.4.2	Graph Transformation	182
9.5	Summary and Outlook	182

10 Towards Hybrid Transformation Modeling 185

10.1	The Need for Hybrid Transformation Models	186
10.2	Balancing between Operational and Declarative	186
10.2.1	Story Diagrams with OCL: too low-level	187
10.2.2	Triple Graph Grammars: too generic	187
10.2.2.1	Declarative Transformations: Graph Grammars	188
10.2.2.2	Application to Example	188
10.2.2.3	Problem Identification	191
10.3	A Hybrid Model Transformation Language	192
10.3.1	Control Flow	192
10.3.1.1	Controlled Triple Graph Grammars	193
10.3.1.2	Control flow alternative	196
10.3.2	Reflection	197
10.3.2.1	Reflective Triple Graph Grammars	197
10.3.2.2	Reflection Alternative	198
10.4	Related Work	199
10.5	Summary and Outlook	200

11 Summary and Conclusions 203

11.1	Original Motivation	203
11.2	Solutions	204
11.3	Final Conclusion	206

Background

Chapter 1

Introduction

1.1 Context

Integrating packages and applications written in a variety of languages is a major challenge for the software engineering community [99]. Luckily, programming languages and their associated libraries have gone a long way. Thanks to modern integration platforms such as J2EE [170] one can, for example, connect mainstream ERP packages such as SAP [222] with legacy applications written in Cobol or a reasoning engine written in Prolog [237, 56]. Unfortunately, this flexibility comes at the cost of high complexity. Managing entities and processes that are scattered over many applications is problematic when there is a lack of documentation on how these applications relate to the overall requirements. Even when such documentation is in place, it is often unfeasible to update all sources and documentation affected by a particular business change without advanced tool support.

1.1.1 Model-Driven Engineering

This thesis contributes to the field of model-driven engineering (MDE [81]), an approach to software engineering that aims to tackle the aforementioned complexity by treating software artifacts as models that can be constructed, merged with other models and evolved into new (versions of) models by applying transformations. In its most general meaning, a model is defined as *a simplified representation of a part of the world named the system* [233]. A model is useful if it helps to gain a better understanding of the system.

In an engineering context, a model is *useful* if it helps deciding the appropriate actions that need to be taken to reach and maintain the system's goal. The goal of software is to automate some tasks in the real world. Models of software requirements, structure and behavior at different levels of abstraction help all stakeholders deciding how this goal should be accomplished and maintained. According to this definition, source code is a model too since it is a simplified representation of the lower-level machine structures and operations that are required to automate the tasks in the real world. Moreover, *correct* source code is a very useful model since it tells the machine what actions need to be taken to maintain the system's goal. Design representations of the source code are useful models if they make the source code more understandable.

A key property of models in an MDE context is their ability to be transformed automatically. When building modeling tools, one needs to model the structure and well-formedness rules of the language in which the models are expressed. Such models are called metamodels [82]. Having a precise metamodel is a prerequisite for performing automated model transformations.

A modeling language can be modeled by different metamodels. One metamodel may simplify the language by only representing semantical information while another one may only represent syntactical information. The former kind of (meta-)model is often called an Abstract Syntax Graph (ASG) while the latter one is often called a concrete syntax graph [73].

1.1.2 Standardization

MDE is embraced by various organizations and companies, including OMG, IBM and Microsoft. MDE encompasses a wide variety of technically different, yet conceptually very similar approaches, including Microsoft's software factories [107], Vanderbilt's Model-Integrated Computing (MIC [144]), McGill's Computer Automated Multi-Paradigm Modeling (CAM-PaM [173]) and other individual approaches. There is also a wide variety of tools available, such as the Eclipse Generative Model Transformer (GMT) framework, the Generic Modeling Environment (GME) and many more. For a detailed and up-to-date tool overview, the MDE community website can be consulted [81].

While this diversity fosters competition and ensures the sustainability of the emerging paradigm, there is a rising need for MDE standards. Without such standards, precious time is wasted on miscommunication between practitioners of different MDE sub-communities and the construction, maintenance and integration of tool fragments that would better be shared. Moreover, the diversity between the different MDE approaches makes teaching the paradigm quite challenging, which in the end hinders the industrial adoption of new techniques.

This need for standards can be compared with the need for the Unified Modeling Language (UML) in the mid-1990s, when the number of modeling languages had increased from ten to more than fifty [32]. The first version of the UML was standardized by the Object Management Group (OMG) in 1997. While UML has been relatively successful as a standard notation for modeling OO software, there is less consensus about the semantical relation between the different diagram types [278, 172]. Additionally, a significant amount of research has been invested in the development of the Object Constraint Language (OCL [198]) to supplement the originally informal diagram types of the UML with a means to model precise business constraints unambiguously [45, 9, 44].

In 2001, the Object Management Group introduced the Model-Driven Architecture (MDA) as a set of standards for MDE [192]. UML was still proposed to be the core modeling language. However, instead of focusing on one fixed semantics, UML models were supposed to be enriched with domain-specific semantical information. Semantical constraints between different diagram types could be enforced by evaluating domain specific well-formedness rules (WFRs) expressed in OCL. The structure of the UML and other languages could be defined in a variant of UML class diagrams, called the Meta Object Facility (MOF [193]). By relying on a language binding such as the Java Metadata Interface (JMI [168]), the domain-specific models could then be queried and transformed in a standard manner. Additionally, queries and transformations of models serialized according to the XML Metadata Interface (XMI [199]) standard could be implemented with general purpose XML technology such as XSLT too [137]. Still, these partial standards did not satisfy industry requirements leading to

the widespread use of proprietary, often ad-hoc, approaches to model transformation [54, 50].

In 2002, the OMG issued a request for proposals to standardize this so-called domain of *Queries, Views and Transformations* (QVT [194]). In late 2005, the OMG proposed a standard consisting of three languages for model transformation [197]. While these languages were based on quite some popular research languages, the still emerging QVT standard suffers from problems that are similar to those that have slowed down the adoption of the UML standard. Therefore, one may expect that the lessons learned from the UML standardization effort are taken into account to ease the adoption of the QVT standard. Going one step further, one may even want to specialize the UML for modeling model transformations: after all, since modeling languages are already modeled in an object-oriented manner using MOF, it seems quite natural to consider model transformations as object-oriented behavior.

Surprisingly, the relation between the UML standard and the QVT standard is minimal. While all three QVT languages use a fragment of the OCL to express paths across model elements, the *design-by-contract* constructs (invariants, pre- and postconditions) that dominate most OCL specifications for business models are supposed to be irrelevant for QVT. Furthermore, the QVT standard does not rely on UML's visual diagram types for modeling behavior. Instead, new language concepts such as *template expressions*, *areas*, *domains* and *rules* are introduced. This creates the impression that model transformation is fundamentally different from object-oriented (meta-)programming. In that case, the mature knowledge in object-oriented analysis, design, testing and restructuring would need to be revisited to match the new paradigm introduced by the standard model transformation languages.

1.1.3 Summary of Contributions

This thesis investigates how model transformations can be developed in a model-driven manner. The thesis shows that the modeling of model transformations does not require a complete new set of modeling (or programming) languages. More specifically, it shows that a variety of model transformation solutions can be modeled precisely in an object-oriented manner: several model refactoring, refinement and model synchronization case study solutions have been tackled using object-oriented modeling languages.

The proposed modeling paradigm is based on design by contract and graph transformation. The following techniques have been contributed to these domains:

- modeling transformation behavior as controlled graph transformations,
- exchanging transformation models between different UML tools,
- developing transformation models that are independent of the target modeling platform,
- decomposing the complexity of large rewrite rules by means of views,
- modeling the copying of subgraphs within (or across) models,
- extending transformation languages modularly, by means of higher order transformations,
- using design by contract in the context of refactoring and model synchronization,
- integrating declarative and operational languages for modeling the latter kind of transformations.

Moreover, the thesis is supported by a taxonomy of the domain of model transformation. This enables one to put the contributed techniques in a broader context than that of the concrete case studies of this thesis. Moreover, other techniques can be related to the proposed techniques more easily.

To facilitate the industrial adoption of the proposed modeling techniques, this thesis uses the most popular diagram types of the UML (class and activity diagrams [65]) where possible. By relying on the mature fundamentals of graph transformation languages, the proposed techniques do not suffer from the ambiguous semantics that have often been associated with the UML [58, 236, 235, 114].

In fact, the semantics of the OCL and graph transformation language used within this thesis has been the subject of much more research than that of the preliminary QVT languages [215, 218, 34, 72, 76, 75, 53]. Therefore, the proposed approach to modeling transformations not only relies on UML fragments that are widely known from a syntactical point of view: instead, the thesis also relies on semantical concepts that should not require an unfeasible amount of training for generally educated software engineers.

The proposed techniques are supported by a tool that compiles the transformation models into Java code. As such, the standard, human-oriented, specifications are also machine-executable on the low-level interfaces of an MDA (i.e., MOF) compliant model repository. This tool relies on template technology that has already been industrially applied for generating code from application models.

Additionally, this thesis shows how new transformation language constructs can be normalized into existing ones by means of a higher order transformation. Among other things, this approach ensures that transformation models defined with the *copy operator* defined in this thesis can easily be executed on transformation tools that have been developed by others.

This thesis does not aim to replace the emerging QVT standard. Instead, it complements the standard in the following scenario's:

- transformations that have been *modeled* with the techniques from this thesis can still be *implemented* with a language from the QVT standard,
- QVT based transformation specifications can be mapped on the proposed kind of transformation models.

Especially the latter approach is an interesting direction for future work.

The remainder of this introductory chapter introduces the reader to the standards for Model-Driven Engineering in more detail. Therefore, experts in the field of Model-Driven Engineering are invited to skip the remainder of this chapter and move to Chapter 2 directly. Other readers may find the rest of this chapter a useful introduction to the field: Section 1.2 clarifies how standard modeling languages can be specialized for domain-specific purposes using the UML and MOF. Section 1.3 focuses on existing approaches to represent model transformations in a standard-compliant manner. Section 1.3.4 discusses a particular approach in more detail. Finally, Section 1.4 summarizes this introductory chapter and presents an outlook for the upcoming chapters.

1.2 Model Driven Architecture

This section introduces the reader to the key MDA standards required to understand the contributions of this thesis. Subsections 1.2.1 and 1.2.2 discuss two complementary approaches

to language definition. Subsection 1.2.1 presents language profiles as a mechanism to specialize general purpose languages into more domain specific ones. Subsection 1.2.2 illustrates the alternative *MOF* approach to language definition. Finally, subsection 1.2.3 discusses the MDA standards related to model transformation.

1.2.1 The UML as a Framework for Domain Specific Visual Languages

The UML has evolved from a toolkit consisting of nine diagram types in UML 1 to an elaborate set of thirteen diagram types in UML 2. The relationships between different diagram types are encoded by associations and inheritance links in the UML metamodel on the one hand and by additional well-formedness rules on the other hand. This allows one to choose from a large library of visual elements when looking for an appropriate representation of a particular aspect of a system. Still, the UML can, and should, not be extended to include a unique visual element for each potential aspect of any conceivable system. Instead, the UML diagram types should be usable in the context of different methodologies and for different types of systems. In fact, the UML should be extensible to ensure its applicability even for system types that were not known by the designers of the latest UML standard.

As long as one of the UML diagram types provides a reasonable visualization of a system's aspect, that system can be modeled without an extension of the standard UML metamodel. Still, the semantics of a language construct can become ambiguous when it is used for modeling different types of software or system artifacts. In that case, a set of UML language constructs can be extended and packaged along with application models without losing interoperability with generic UML tools. Even stronger: after importing these packages, called "profiles", into such generic tools the functionality of their editors is extended with some default editing behavior that is sufficient to manipulate models conforming to the apparent extension of the UML.

1.2.1.1 UML Profile Application

To illustrate both the application and the definition of such a profile, consider the following models of a "Meeting Scheduler" application. The application is based on the *Meeting Scheduler* problem statement that was proposed by Van Lamsweerde et al. [141] as a benchmark for requirements elicitation and software specification techniques [83]. Subsequent chapters will present more example models of that application.

Figure. 1.1 displays that each meeting attendee can specify (1) in what time intervals he/she cannot attend, (2) in what time intervals he/she would prefer to attend and (3) the flexibility (or priority) for (2). Allowing the use of association classes (such as *Flexibility*) allows a "conceptual data modeler" to represent the relationship for date preferences directly between its major participants (*Attendee* and *Time Interval*) without putting too much focus on the auxiliary *Flexibility* class. The *acceptableLimit* and *cancelLimit* attributes of class *Schedule* serve as deadline mechanisms in the algorithm for proposing an optimal meeting location and date. To formalize that this diagram is part of a "conceptual data model", its containing *Model* is marked (or "stereotyped") with a model element that is part of a profile package. The name of that model element is "Conceptual Model" while its type is the UML metaclass called *Stereotype*. By simply adding this flag, all classes, associations and association classes acquire a meaning that is more specific than that of UML classes and associations in general.

Figure 1.1: Data structure for representing the date preferences of meeting attendees.

Figure 1.2: Physical data model of “date preferences” information.

Figure 1.2 shows the physical data model corresponding to the conceptual model from Figure 1.1. The model conforms to a profile for physical data modeling [255] based on the work of Ambler [232]. The mapping from classes and associations with association classes to tables, associative tables, keys and foreign keys corresponds to the mapping from entity sets and relations with attributes to relational schemata as covered in introductory database course books [125]: each class from the logical data model is mapped to a table. All attributes are included as columns, along with extra columns for realizing the associations between the classes. The physical data model contains explicit nodes for keys and foreign keys such that all relations are defined precisely.

The associative table *ATTENDEE-TIMEINTERVAL* maps to the association class *Flexibility*. The table maps to two outgoing one-to-many associations. We present this mapping problem to illustrate that there is a consistency relation between conceptual models and physical data models demanding that each class maps to a table and vice versa.

Stepping aside from the semantics of the given meeting scheduler models and focusing on the UML notation, one should note that *Stereotype* elements are applied much more extensively than in Figure 1.1: all nodes of the decorated graph are of type *UmlClass*. However,

several nodes (e.g., *PERSON* and *PERSON_KEY*) should have a different semantics. Moreover, this semantics is much more concrete than the default semantics of *UmlClass*. Therefore, *PERSON* is decorated with the stereotype with name “Table” while *PERSON_KEY* is decorated with the stereotype with name “Key”.

The *Key* construct of the physical data modeling profile illustrates the final concept that is essential for understanding the expressive power of profiles: since keys consist of a number of table columns, the *Key* stereotype is provided with an attribute called “columns”. This attribute, which is called a “tagged value” in the UML, has a list of UML *Attributes* as its type. Therefore, when editing the *TIMEINTERVAL_KEY* element from Figure 1.2, a CASE tool would allow the modeler to change the *FROM* and *TO* values into another list, say {*FROM*}, but the tool would disable the user from adding a *UmlClass* such as *TIMEINTERVAL* to the list.

1.2.1.2 Evaluation

As Figure 1.2 illustrates, language profiles allow one to define domain specific visual modeling languages whose syntax can differ to a reasonable extent from the extended UML diagram types. This flexibility is an essential characteristic for making the UML applicable as a standard, yet versatile, language definition framework. The default editing functionality that was discussed in the context of the *TIMEINTERVAL_KEY* element from Figure 1.2 illustrates that this flexibility is complemented with a reasonable degree of type checking.

Still, type checks are insufficient to realize a completely usable editor for physical data models. Coming back to the checks performed when editing the *columns* tag of the *TIMEINTERVAL_KEY* element, users should be restricted to choosing between the attributes of the *TIMEINTERVAL* element. More generally, the value of the *columns* tag should be restricted to a list whose elements are attributes of the classifier that is reached by the UML Dependency link (the dotted, directed arrow) starting from the *Key* element.

Interestingly, the UML profile mechanism provides a means to enforce such domain specific checks. More specifically, these checks can be enforced by means of an OCL invariant defined in the context of the extended UML metaclass. This feature makes the profile concept a language extension mechanism that is sufficiently powerful as long as all new language constructs can be defined in terms of existing UML metaclasses. As soon as language constructs are needed that cannot be considered as subclasses of existing UML metaclasses, a more powerful technique such as explicit object-oriented metamodeling is needed. Such a technique is discussed in subsection 1.2.2 that covers MOF.

1.2.1.3 UML Profile Definition

This section presents how the UML profile for physical data modeling that is applied in the previous section can be defined.

Figure 1.3 shows a part of the content of the package defining this profile. All elements on the figure are regular UML elements that can be stored in a standard UML model that also contains application elements. The *Physical Data Model* stereotype has no particular attributes (tag definitions). As shown in the context of Figure 1.2, its only purpose is to indicate that the elements contained in a particular UML model are relational database constructs instead of object-oriented constructs such as classes and methods. *Table*, *Associative Table*, *Foreign Key* and *Key* are the stereotypes applied in Figure 1.2. The type of the *columns* tag

Figure 1.3: Part of the UML profile for physical data modeling.

of the *Foreign Key* and *Key* stereotypes is defined as a nonempty list of *Attribute*. The latter is a metaclass from the UML metamodel. The type of the origin tag of stereotype *Key* is *Key Origin*, an enumeration type that is also contained in the profile definition package.

One can imagine an arbitrary number of OCL well-formedness rules on this profile. One possible rule would state that a *Classifier* marked with the *Foreign Key* stereotype should have an outgoing dependency link to a *Classifier* marked with the *Key* stereotype.

1.2.1.4 Language Support for Profiles

This section defines the concept of a language profile in a general manner, to illustrate that it is not a UML specific concept.

*In general, a **profile** can be defined as a collection of language constructs (“stereotypes” and “tagged values”) that can specialize other language constructs.*

*We define a **stereotype** as a model element that models a subtype of an existing metaclass, called the “base class”.*

A subtype relationship between metaclasses needs to be *modeled* when the abstract syntax of the language cannot be changed directly. This may occur when models that are expressed in that language *need to be exchanged with a standard API or data structure*. Although theoretically, extensions to a standard API can be adapted by specializing the standard interfaces or classes and relying on the object-oriented substitution principle and on reflection, practice dictates that today’s modeling tools rely on languages with a fixed abstract syntax.

When the abstract syntax of the modeling language is fixed, extra meta-information needs to be expressed in the language itself. In the context of object-oriented metamodeling, a stereotype is a model element A that is an instance of a metaclass B that has a “name” attribute (instantiated to “A” in this example) and that is associated with the metaclass C part of metamodel D. Although conceptually A is a metaclass that subclasses C, it is impossible to create instances of A in repositories that are based on the types of D. Instead, one can state that a model element E that is an instance of type C conforms to A by adding A to the set of Bs associated with E.

Example UML stereotypes from Figure 1.2 are *Table*, *Foreign Key*, *Key* and *Associative Table*. These model elements (corresponding to “A”) are instances of the UML metaclass

Figure 1.4: MOF diagram of UML metamodel support for profiles.

Stereotype (corresponding to “B”). All four stereotypes are defined on UML metaclass *Classifier* (corresponding to “C”) of metamodel *UML* (corresponding to “D”). *Table*, *Foreign Key*, *Key* and *Associative Table* conceptually subclass *Classifier* but technically they are only associated to *Classifier* through the *base class* attribute of UML metaclass *Stereotype*. Figure 1.4 elaborates on the design of profiles in the UML 1.5 metamodel. The definition of the stereotypes for the physical data modeling profile is visualized on Figure 1.3. On Figure 1.2, *Table* is applied to model element *Person* (corresponding to “E”).

*When generalizing the concept of **tagged values**, they can be defined as a model element that models an attribute of a stereotype.*

Tagged values are needed when the attributes of the base class of a stereotype are not sufficient to model the domain of interest. Example tagged values from Figure 1.2 are *columns* and *constr* which are defined on Figure 1.3. The *columns* tagged value provides the modeler with a new attribute on classes with the *Key* stereotype. For the *TIMEINTERVAL_KEY* key on Figure 1.2, the *columns* tag expresses that both columns (attributes “FROM” and “TO”) of the *TIMEINTERVAL* table define the key of that table. Note that with ordinary UML class elements, one could not refer to the attributes from another class in this way.

Stereotypes and tagged values are distributed in packages, called *profiles*, that can be included as libraries in applications. Since stereotypes and tagged values are model elements in their own right, they have to be formalized in the metamodel of the modeling language being used. Figure 1.4 displays the part of the MOF metamodel of UML responsible for defining and applying profiles. A model element can have zero or more stereotypes associated with it.

The right side of the diagram shows that each stereotype can have a number of contained tag definitions. This enables the composition on the left side of the diagram to be populated: a model element can have many tagged values associated with it. Each such tagged value is either string-based (when *dataValue* is instantiated) or conforms to exactly one tag definition (when the association with end *referenceValue* is populated). A tag can associate a model element with multiple values (such as the *columns* tag on *TIMEINTERVAL_KEY* on Figure 1.2), depending on the *multiplicity* property of the tag definition. The language profile technique is also implemented in Java, where tagged values are called *annotations*.

1.2.2 Meta Object Facility (MOF)

As subsection 1.2.1.2 indicated, some language elements cannot be expressed by means of stereotypes and tagged values. More specifically, when a new language element has no specialization relationship to any existing language element, a new metaclass needs to be introduced. The Meta Object Facility (MOF) provides a standard language to define such new metaclasses. Since this language is used to define new metamodels, it is often called a meta-metamodeling language. Similarly, the metamodel for this language is often called a meta-metamodel.

The MOF language strongly resembles the class diagram fragment of the UML. In fact, from the example given on Figure 1.4, one cannot derive whether UML or MOF syntax is used. Still, the UML and MOF metamodels have historically been separated to enable an evolution of the UML metamodel without affecting that of the MOF.

Keeping the MOF metamodel stable is driven by the large impact a change in the MOF metamodel would have on existing model repositories and XMI files. More specifically, when changing the MOF language, all metamodels defined in that language need to be redefined. In turn, the changes of those metamodels would require all models defined in the corresponding modeling languages to be *redefined* (or at least *migrated*) too. In contrast, a change of the UML metamodel, only requires a migration of existing UML models and does not affect how Entity/Relationship or Java models are stored.

Luckily, the required expressiveness for defining new metamodels is fairly limited. In summary, the core of MOF allows one to define new classes (with attributes and black-box operations) that can be associated with one another, specialize one another and reside in packages that can import other packages. By standardizing more advanced data modeling constructs (such as association inheritance) in MOF 2.0, the OMG introduced several new research and implementation related challenges [5].

1.2.3 MOF Queries, Views and Transformations RFP

The QVT RFP provides an interesting set of requirements for standardizing the domain of model transformation in the context of the MOF and related technologies such as XMI [194, 193, 196]. This subsection highlights some of these requirements while Chapter 2 will discuss the design space of model transformations more completely by presenting a comprehensive taxonomy and a set of success criteria for related tools. However, since the OMG is the only standards body promoting standards for model-driven engineering, the RFP is an interesting starting point.

1.2.3.1 Requirements from the Request For Proposals

The RFP presents both a list of general, yet technical, requirements, that are not specific to model transformation as well as a set of specific model transformation language requirements. The latter requirements are separated further into mandatory and optional requirements.

A majority of the general requirements refers to compatibility with existing modeling standards such as XMI, MOF and OCL. First of all, the RFP calls for a transformation language that specifically reuses OCL for querying models, XMI for the serialization of all artifacts and MOF for the definition of metamodels. Additionally, the RFP generally favors the reuse of

existing standards, with minimal changes. Finally, it is encouraged that standards allow as much implementation flexibility and substitutability.

The most obvious requirements specific to MOF Queries, Views and Transformations are the ability to express model transformations, model queries and model views. More specifically, the RFP mandates that one target model can be generated automatically from a source model. Additionally, the RFP mandates that after such a transformation, changes to the source model can be propagated incrementally to the target model. This “incremental update” feature should be declarative, in the sense that one transformation specification should enforce the generation of a new target model as well as the subsequent manipulations due to changes in the source model without explicitly specifying these two cases.

By default, the RFP assumes that the metamodels of source and target models are distinct. However, the case where these metamodels are the same should also be supported. Finally, support for the case where the target model is just an updated version of the input model is optional. The other optional requirements are the ability to reuse, specialize and parametrize transformation definitions.

Remarkably, the RFP briefly refers to the potential of using the UML action semantics constructs for modeling the behavior of model transformations. However, the UML meta-model fragment for action semantics has primarily been developed as a pivot metamodel for translating between other concrete languages such as SDL, C++ or Java. Consequently, the initiative for standardizing action semantics did not result in a concrete syntax that could be used to model the behavior of model transformations.

Therefore, this thesis relies on another behavioral modeling language whose concrete syntax has been designed specifically for representing complex queries and transformations. As a contribution to this thesis, this language is aligned with the OMG standards in Chapter 5 and extended for raising the level of abstraction for large transformations in Chapter 7.

1.2.3.2 Evaluation of Existing Solution: Java Metadata Interface

The MOF language not only enables one to define how classes from a metamodel are related to one another: it also enables one to specify what operations should be provided by such classes. In that sense, one can realize queries and views on models by providing such operations with the appropriate behavior. One way to implement the behavior of such operations is to rely on a general purpose programming language.

To use this model in a particular programming language, one needs to agree how elements from that model can be created, read, updated or deleted. The Java Metadata Interface (JMI) defines how models and their elements can be accessed from Java code [168].

More specifically, JMI defines one or more Java entities for each element of the MOF model, thus introducing a standard API for querying and updating metamodels. For example, a MOF class is mapped to two Java interfaces: one “factory” (or “class proxy”) interface for constructing objects and one “instance” interface for manipulating them. An equivalent mapping rule is defined for MOF associations.

This mapping from the MOF model to Java sources is model-driven since it is supported by a code generator that produces source files from metamodels conforming to the MOF model. Since the MOF model conforms to itself, the two Java interfaces are generated for the classes “MOF class”, “MOF attribute”, etc. too. Note however that the MOF model is supposed to be stable. Therefore, the main applicability of a JMI code generator lies in the production of Java sources from new metaclasses and their associations: by applying the generic mapping

to other metamodels than MOF, a metamodel-specific set of interfaces is obtained, through which any instance of this metamodel can be accessed and manipulated in a uniform manner. In the case of UML, for example, these interfaces can be used to add a new UML class to a model of a class diagram, or find an existing UML association and delete it.

In addition to this generative approach, the MOF standard also provides a reflective API for accessing models. The reflective API is equivalent in power, yet contains metamodel-specific information in string arguments instead of strongly typed object instances.

Although the JMI standard is a promising basis at the execution level of model transformations, it has not been designed for *modeling* model transformations. Therefore, this thesis investigates how more human-oriented models can be mapped automatically to JMI compliant source code.

1.3 Towards Standards for Modeling Transformations

This section provides an overview of the research that has already been conducted in the field of MDA compliant transformation modeling, or, more generally: in the field of standard compliant model transformation definition. In particular, the section discusses existing techniques that rely on contract based languages such as the OCL and visual modeling languages such as the UML. Subsection 1.3.1 summarizes the research that has lead to OCL based action languages such as *Kermeta* and OMG's *Imperative OCL*. Subsection 1.3.2 summarizes other OCL based research that revisited the design by contract fundamentals in a transformation modeling context. Subsection 1.3.3 presents how OCL has been applied in a consistency maintenance context. Subsection 1.3.4 presents research on how the actual behavior of a transformation can be modeled visually. Finally, subsection 1.3.5 briefly summarizes the contributions of the emerging QVT standard.

1.3.1 Jézéquel's OCL Actions

The IRISA team, under the supervision of Jézéquel at the university of Rennes, proposed to extend the OCL with side-effect constructs [243, 207]. The idea is to extend the OCL *pre*, *post*, and *inv* constructs with an *action* construct that allows one to describe the side-effects that need to take place within a method body.

While it is not clearly illustrated why the proposed assignment and method call operators enable one to specify behavior in a more declarative way than the operators found in mainstream programming languages such as Java or C#, this approach does avoid the need to specify the model queries part of the pre- and post-conditions in another language than those part of the method body. Moreover, the OCL action language hides the complexity of the underlying repository framework: the creation of a new model element is realized by a straightforward constructor call instead of by a call to a more complex factory method from a JMI proxy.

Still, the use of OCL as an action language suffers from severe limitations both from a syntactical and from a semantical point of view. Syntactically, several aspects of a model transformations can benefit from visual language constructs. The advantages of a visual language are covered elaborately within Chapters 4 and 6. From a semantical point of view, the proposed OCL extensions lack a means to model incrementally updating transformations (as defined in subsection 1.2.3), which is not only a mandatory requirement of the QVT RFP but

which will also be found particularly useful in subsequent chapters of this thesis. Similarly, the proposed extensions do not provide a means to model bidirectional transformations in a symmetrical manner. Again, subsequent chapters will illustrate this leads to low-level transformation specifications. Therefore, the extensions proposed by Jézéquel et al. are defined at an intermediate abstraction level between the APIs of model repositories and higher level languages for modeling model transformations.

Finally, it should be noted that the work on an OCL based action language has been elaborated further within the QVT standardization process. Although a detailed comparison is outside the scope of this thesis, one can easily observe that more care has been taken not to change the semantics of existing OCL constructs: instead of allowing side-effects within the body of an OCL *forAll* expression [207], the OMG’s *ImperativeOCL* extension to conventional (i.e., side-effect-free) OCL provides separate loop constructs for side-effects and preserves the mathematical semantics of the *forAll* construct.

1.3.2 Cariou’s OCL Transformation Contracts

In contrast to the work discussed in the previous section, Cariou et al. aimed to model the behavior of model transformations in standard UML and OCL, without relying on extensions for expressing side-effects, and with sufficient freedom to implement the actual transformation in any transformation language [39, 40].

Although this thesis also aims to ease the adoptability and portability of model transformations by relying on industry standards where possible, the following essential differences should be noted: first of all, Cariou’s work is heavily influenced by a very specific set of model transformations. More specifically, its focus lies on the transformation of only one source model into only one target model, where the metamodels of source and target models are specializations of the UML metamodel. In fact, [39] focuses on transformations that gradually evolve a UML model conforming to one profile into a new version of that model that conforms to another profile. In contrast, this thesis proposes techniques and tool prototypes that are applicable on any number of models conforming to any MOF metamodel. Secondly, this thesis proposes an executable modeling language for expressing the actual behavior of a model transformation. Although the generated implementations can be substituted by code originating from other transformation tools, the proposed framework does not require such tools to realize an executable result. Finally, Cariou et al. deviate from the UML and OCL standards for realizing goals that have been accomplished in a standard-compliant manner by this thesis.

To understand the limitations of Cariou’s standard OCL approach, consider the following fragment taken from [40]:

```
context Client::proxyAddition()
post: -- constraints on the target model and source model elements (with the
      -- @pre OCL construction)
```

The first fundamental design choice is that transformation methods are defined within the context of the metaclass of the UML elements that need to be transformed from the source model to the target model. Note that although *Client* is not part of the UML metamodel, Cariou intends to express a constraint on that type of input element.

Secondly, note that elements from the source model are referred to by means of the *@pre* construct. This is necessary when the state of the input model is changed by the transformation

(operation). Although this may make sense in the context of model synchronization, it is often abused for other purposes tacitly. For example, consider the OCL code in more detail:

```
context Client::proxyAddition()
post: self.dependsOn-> forAll ( i |
  let proxy = self.newProxy() in
  let server = i@pre.isImplementedBy in
  -- the 'client@pre' is equals to self, no variable is required
  server.isTypeOf(Server) and
  proxy.implements-> includes(i) and
  proxy.hasClassRefWith(self) and
  proxy.hasClassRefWith(server)
)
```

The applicability of this code is severely limited since source elements are updated destructively into target elements. More specifically, elements of type *proxy* are inserted directly in the source model. Consequently, at the end of the transformation process, a model without such implementation-related elements will no longer be available. This is undesirable since it hinders traceability. Although the use of the *@pre* construct is not fundamentally incompatible with a model-driven stack of models that are related by traceability links, the construct should be used with care. The taxonomy presented in Chapter 2 will provide a more general discussion of such design consequences.

In an attempt to generalize their approach to transformations between models conforming to different metamodels than that of the UML, Cariou et al. propose some non-standard extensions to the OCL [40]. For example, they propose to attach constraints to packages and diagrams instead of restricting the definition of constraints to a design by contract context (i.e., invariants, pre-, and post-conditions within the scope of a Classifier). This thesis will illustrate that by applying standard MOF extension techniques, there is no need for these proprietary and unconventional extensions. More specifically, Chapter 9 illustrates how transformation contracts can be defined in the scope of classifiers that are part of a metamodel that imports the source and target metamodels.

As another example extension, Cariou et al. claim that standard OCL does not enable one to reason about mappings between model elements. This thesis will illustrate in Chapter 9 that the OCL and MOF standards do enable one to achieve that goal by means of traceability elements that are stored within a model that *clusters* a standard UML model. Technical details on the topic of model clustering are not included in the text but can be found in the MOF specification [193].

In summary, Cariou et al. motivated that the use of OCL transformation contracts is a relevant research topic. However, they also illustrated this topic introduces challenging issues that require further investigation.

1.3.3 Akehurst's OCL Relations

David H. Akehurst has investigated the use of OCL as a model mapping language in several iterations and has probably influenced UML related transformation research and standards several times. This subsection presents his two most representative approaches published in the research literature. In 1.3.3.1, his standardization of a bidirectional mapping operator is presented while 1.3.3.2 provides an overview of his more powerful pattern for modeling mathematical relations in standard OCL.

Figure 1.5: Standard class structure for representing mapped elements [3].

1.3.3.1 Standardization of a Mapping Operator

The first significant contribution of Akehurst discussed in this section is taken from his Ph. D. thesis [3]. By relying on standard mechanisms such as specialization and template binding, it is shown how a special mapping operator ($\leftrightarrow$) can be mapped to standard OCL.

This operator is specialized for modeling relations between two domains. To this purpose a class-based formalization of mathematical pairs, Cartesian products and bijective mappings is presented.

Figure 1.5, taken from [3], shows the standard class structure that implicitly follows from the statement that metaclass X is $\ll mapped to \gg$ metaclass Y (i.e., “ $X \leftrightarrow Y$ ”). Using standard OCL invariants on the *Pair*, *CartesianProduct* and *BjMapping* classes, expressions stating that two objects, two tuples, or two sequences are $\ll mapped to \gg$ one another are given a precise and standard-compliant meaning. In summary, a mapping is formalized as follows: when a metaclass X is mapped to a metaclass Y , there should be exactly one instance of *BjMapping* that satisfies the constraint $C(x,y)$ of the mapping. Since *BjMapping* represents the unique bijective mapping between all instances of X and Y , each $C(x,y)$ associated with a “ $X \leftrightarrow Y$ ”-expression globally defines how instances of X should be mapped to instances of Y .

Figure 1.6 shows an example mapping rule from the domain of “directed graphs” to the domain of “trees”, taken from [3]. This example expresses that an *Edge* element from the former domain should be mapped to two *TNode* elements from the later domain where one *TNode* instance should be contained in the “subnodes” collection of the other *TNode* instance. This corresponds to the intuitive idea of mapping a directed edge to a parent-child structure in a tree. It illustrates several advantages as well as disadvantages of the mapping approach.

As the first illustrated advantage, the example applies the compact syntax for expressing one-to-many mappings. More specifically, the constraint should be interpreted as:

```

let e = self.fst in
let tn1 = self.snd.at(0) in

```

Figure 1.6: Example mapping specification in special syntax [3].

```

let tn2= self.snd.at(1) in
tn1.oclType= TNode and tn2.oclType= TNode and
BjMapping<Vertex,TNode>.allInstances()->exists(m|m.fst==e.start and m.snd==tn1) and
BjMapping<Vertex,TNode>.allInstances()->exists(m|m.fst==e.finish and m.snd==tn2) and
tn1.subnodes->includes(tn2)
  
```

Secondly, the transformation writer does not have to provide an explicit class definition for realizing traceability links between the two domains. Instead, the links are realized by means of the generic library classes discussed before. In the derived specification presented above, it can be seen how the instances of the template bindings of the library *BjMapping* class provide a generic traceability structure. No MOF classes with associations to *Edge* and *Pair<TNode,TNode>* have to be defined explicitly.

The example also illustrates three disadvantages of the mapping approach. First of all, the second advantage discussed above obviously comes at the cost of more complex (or at least more implicit) expressions when navigating across traceability links.

Secondly, the presented approach lacks a scoping mechanism. More specifically, it may be desirable to express that only directed edges within a particular subgraph need to be mapped to a tree. Similarly, there may be *TNode* pairs in the tree domain that have no mapping to a directed edge in the graph domain. Instead, such a pair may represent the two root nodes of trees that are derived from two directed subgraphs that require a tree analysis. In these cases, the OCL constraint presented above would fail since more than one instance of the same kind of *BjMapping* binding would be connected to the same *TNode* instances.

Finally, the approach suffers from a general lack of generality since only bijective mappings are supported. In practice however, several mapping problems do not call for a bijective solution. For example, when mapping all objects of a system snapshot onto their classes, all classes will have a type but not all types need to be active within the system. Therefore, the mapping from classes to objects is not surjective and consequently not bijective.

These problems are overcome in the approach presented in the next section.

1.3.3.2 Pattern for Modeling Mathematical Relations in the UML

Together with Kent and Patrascioiu, Akehurst generalized the above approach for modeling global bijective constraints into a pattern-based approach to modeling mathematical relations in general [2].

The key to this increased generality is that the domain and range of the relations are no longer realized through template bindings from the *BjMapping* class with its fixed invariant. Instead, relations specialize a generic *Relation* class that provides a set of constraints as OCL helper operations. In the class diagram shown in Figure 1.7, taken from [2], these helper

Figure 1.7: Library of *Relation* and *Pair* metaclasses [2].

operations are listed in the context of the abstract *Relation* class. Concrete relations *can* assert this operation from an invariant definition, *if* desirable, but *not* by default.

The helper operations are defined in standard OCL, as illustrated by the following fragment from [2]:

```

context Relation def:
  let isFunctional() : Boolean =
    domain->forall(x |
      self.pairs()->select(p | x = p.domainElement())->size < 2
    )
  ...
  let isInjection() : Boolean =
    isFunctional() and isInverseFunctional()
  let isBijection() : Boolean =
    isInjection() and isOnto() and isTotal()

```

As a second improvement to the previously discussed approach, relations are bound to a precisely defined scope: while in the previous approach, the domain and target of a relation consisted of the complete set instances of the source and target types respectively, relations are provided with explicit *domain* and *range* associations to the source and target types respectively. These associations are shown on the class diagram in Figure 1.7 too.

As a supplementary scope mechanism, a *Relation* instance is defined within the context of a parent *Pair* instance, or in the context of another *Relation* instance. Consider for example on Figure 1.8 a class diagram, taken from [2], that models the hierarchy of mapping classes that manage the consistency between classes and object snapshots. In the middle of Figure 1.8, the *ClassObject* metaclass is displayed. This class represents a mapping from one class to one object of its type. Since this class implements the abstract *Pair* class, it needs to implement the *domainElement* and *imageElement* operations to indicate which two elements are mapped by the concrete pair. For *ClassObject*, these operations simply return the *class* and object *link* values respectively.

Figure 1.8: Scoping of pairs in relations that are nested within a *context* pair [2].

The *ClassObject* “pair” class is aggregated by the *ClassRelObject* “relation” class that represents the complete set of class-to-object mappings. As a concrete relation, its associations with ends *domain* and *range* are constrained in terms of the content of the *package* and *snapshot* elements that are mapped by *PackageSnapshot* pair at a higher level within the scoped relation hierarchy:

```
context PackageSnapshot inv dom_ran_of_classRelObject :
  classRelObject . domain = package . class and
  classRelObject . range = snapshot . object
```

Once this invariant has ensured the *domain* and *range* sets have been given a concrete content, the *pairs* of the relation can be constrained “quantitatively” in terms of their cardinality as well as “qualitatively” in terms of the attributes of the mapped elements: the invariant *classRelObject_properties* states that all objects should be mapped to exactly one class, while the *match_lhs_rhs* invariant of the *ClassObject* class states that *Class* and *Object* elements that are mapped to one another by means of the *ClassObject* pair class should have the same name:

```
context PackageSnapshot inv classRelObject_properties :
  classRelObject.isOnto() and
  classRelObject.isInverseFunctional()
...
context ClassObject inv match_lhs_rhs :
  class . name = object . class
```

Coming back to the scoping of mapping rules, it is important to note that *Classes* and *Objects* that are mapped by other pairs may be subject to other constraints while elements that are not related are not constrained at all. To complete the discussion of the scoping issue, remark how the mapping of packages to snapshot is bound to a scope:

```
context PackageSnapshot inv dom_ran_of_packageRelSnapshot :
  packageRelSnapshot . domain = package . package and
  packageRelSnapshot . range = snapshot . snapshot
```

While this constraint does ensure that packages that are recursively contained within a mapped package, are mapped too, it does not constrain packages that reside in a part of the model that is unrelated to the mapping to snapshots. A similar remark can be made in the context of the “snapshot domain”: only snapshots that are contained within already mapped snapshots should be mapped to packages. Although it is not explicitly mentioned within [2], this scoping mechanism strongly resembles that of Triple Graph Grammars, a formalism that is discussed in Chapter 10. From that domain, it is known that to apply the set of mapping rules for a particular application, one either needs an “axiom rule” that explicitly asserts which top-level element from one domain needs to be mapped to which top-level element from the other domain, or a user needs to create a mapping pair (i.e., “traceability link”, or “correspondence link”) between such elements manually.

Coming to the strengths and weaknesses of the relational pattern presented in this section, its primary strength consists of its support for the modeling of a wide variety of model transformation contracts without the need for extensions to the object-oriented paradigm. The pattern is not only presented as a style of modeling. Instead, a reusable of metaclasses is provided, along with a library of OCL helper operations that supports the modeling of the most commonly known mathematical relations such as functional relations, injections and surjections.

As a second advantage, the approach is supported by an implementation in the Kent Modeling Framework which does not have any conceptual mismatches with a MOF based architecture. The OCL expressions do not rely on non-standard constructs either. Finally, the scoping problems discussed in the previous section have been solved.

Unfortunately, the pattern based approach presented in this section has some drawbacks and limitations too. As a first disadvantage, one could argue that the explicit definition of *Relation* and *Pair* classes is more time consuming than the previously presented approach. However, this does not outweigh the semantical clarity that follows from that explicitness. If the definition of the mapping classes is considered to be too expensive, one may generate them from a more abstract description.

As a second disadvantage, the invariants discussed in this section are spread over different metaclasses. While this design is required for capturing the desired semantics of *bidirectional mappings* with a minimal amount of code duplication, other kinds of transformation problems can be modeled by constraints that capture the relations between several metaclasses in the context of only one transformation class. Therefore, the taxonomy from Chapter 2 will introduce the reader to the different kinds of model transformations and subsequent chapters will illustrate that different problems require different modeling approaches.

As a final disadvantage, it can be noted that the relation between this OCL based approach and other, more mature, approaches such as Triple Graph Grammars, has never been investigated in much detail. In fact, it appears that the OCL and graph transformation communities have had very little interaction historically.

As a first limitation of the approach, it is remarkable that no formalism has been proposed to model the actual transformations. Akehurst et al. have implemented these transformations in Java and do remark that more of this manually written code should be generated but have not completed this line of research.

As a second limitation, the proposed modeling pattern does not model sufficient information explicitly to generate a set of default model transformation implementations from the pure OCL contracts. In the meanwhile, the emerging QVT relational standard does include a syntax that is sufficiently complete for this purpose. It is based on the Triple Graph Grammar formalism that is mentioned several times already by this text and that will be presented in more detail in Chapter 10.

Still, this second limitation does not make the proposed pattern irrelevant since its design-by-contract semantics is much more conventional and therefore better understood than that of the more declarative formalisms. Therefore, a promising line of future work is to map the QVT relational or Triple Graph Grammar syntax onto models conforming to the pattern discussed in this section.

1.3.4 Controlled Graph Transformation, Story Diagrams

While the previous subsections highlighted early attempts to formalize contracts for model transformations in a standard constraint language, this section introduces the reader to Story Diagrams as a natural candidate for modeling the behavior of these transformations explicitly. Modeling such behavior explicitly is useful in cases where structuring the transformation contract in a form from which imperative behavior can be derived automatically results in OCL specifications that are too difficult to maintain. Moreover, some transformations require more specific behavior than what can be generated by generic engines.

Story Diagrams is today's most popular language for controlled graph transformation. Controlled graph transformation gained industrial credibility in the nineties, thanks to the application of the Progres language (and tool) within industrial tool integration projects [181]. After working within this team [283, 231], Albert Zündorf defined Story Diagrams as a UML based syntax for graph transformation and supported it by a Java based tool called Fujaba [100, 200].

While Zündorf and others illustrated that Story Diagrams are a promising language for application modeling in general, embedded in the Story Driven Modeling (SDM) process [281, 95, 63], graph transformation has been historically successful in CASE tool integration projects. Within this context, industrial applications indicated the need to execute consistency checks and model transformations directly on the APIs of the integrated tools while this was hard to achieve with Progres [122]. Therefore, this thesis builds upon Story Diagrams as a promising language for standardizing model transformations.

For a more elaborate introduction to the development process of Story Driven Modeling and the related languages, the interested reader is referred to a technical report [256]. However, to ensure this thesis remains self-contained, this section presents some of those concepts of Story Diagrams that are used throughout the upcoming chapters.

This introduction should make the language understandable to readers that are already familiar with mainstream object-oriented programming languages such as Java. Subsection 1.3.4.1 introduces the reader to primitive graph rewriting rules while subsection 1.3.4.2 explains why the concrete syntax of Story Diagram slightly deviates from the mainstream rewriting syntax. Subsection 1.3.4.3 covers the syntactical constructs to control the execution of different rewrite rules explicitly. Finally, subsection 1.3.4.4 briefly evaluates the strengths and limitations of Story Diagrams and the related process.

1.3.4.1 Graph Transformation Basics: Left- and Right-hand Sides

The most primitive and probably most widely known concepts of graph transformation systems are a *rule* (or *production*), a *match* (or *occurrence*) and a *transformation step* (or *direct derivation*). First of all, a graph transformation rule consists of a graph pattern, that needs to be looked up in a host graph. This pattern is often called the left-hand side of the rule because a large number of graph transformation languages visualize this pattern on the left part of the rule. Secondly, a rule contains a graph pattern that should be inserted into the host graph after the first pattern has been found. For similar reasons, this pattern is often called the right-hand side of the rule.

The precise semantics for matching the pattern on the left-hand side has been formalized in terms of category theoretical concepts. More specifically, one has defined what kind of morphisms need to exist between the nodes and edges in the left-hand side of a rule and the nodes of a match in a host graph. Several variants have been proposed, which differ from each other by:

1. whether or not different nodes in the left-hand side are allowed to be mapped to one node in the host graph, and
2. whether or not a node can be deleted when such a step would produce edges related to only one node (i.e., *dangling edges*).

The first difference relates to the *injectivity* of matches. In this thesis, matches are *non-injective* by default but an additional "application condition" can be used to ensure that a

particular rule only matches injectively. The official Fujaba version of Story Diagrams supports injective matching by default [100].

The second difference distinguishes the “double pushout approach” [48], that statically enforces the explicit “gluing condition” to ensure all node and edge morphisms are homomorphic, from the “single pushout approach” [71], that automatically deletes dangling edges when they happen to be produced at rule execution time. This thesis relies on the single pushout semantics but this design choice does not have impact on the thesis contributions.

The following algorithmic steps, based on [142], clarify in a simplified way how a graph transformation rule can be applied to a host graph G_{host} :

1. Identify the left-hand side G_{LHS} within the host graph G_{host} . Note that the fulfillment of specific *application conditions* must be checked. These conditions can be *negative*, in which case a pattern will only be matched if the specified structure is not present in the host graph.
2. Delete from G_{host} , each element (i.e., node or edge) e_{lHost} , that corresponds to an element e_{lRule} from G_{LHS} when e_{lRule} does not occur in the right-hand side G_{RHS} .
3. Create a graph element e_{rHost} in G_{host} for each element e_{rRule} that is part of G_{RHS} but is not part of the left-hand side G_{LHS} .
4. For each node n_{rRule} in G_{RHS} that carries “attribute value assignments”, update the related attribute values of the node n_{rHost} in G_{host} that corresponds to n_{rRule} .

1.3.4.2 Concrete Syntax: Merging LHS and RHS

Several graph transformation languages (such as Progres [228]) display the left- and right-hand sides separately. However, this has some disadvantages.

First of all, the transformation writer needs to label all elements from both sides to indicate what elements occur on both sides. Secondly, when reading a transformation rule, one needs to investigate these labels carefully to find out what nodes are part of the left-hand side without occurring on the right-hand side and vice versa. Finally, in order to compensate the previous disadvantage, one needs to maintain the same layout for all elements that occur both on the left- and right-hand side of the rule.

Therefore, other languages, and Story Diagrams in specific, eliminate the duplicate specification of elements occurring on both sides of the rule such that the differences between the two sides becomes more visible. Nodes and edges marked with the $\ll destroy \gg$ stereotype appear only on the left-hand side of the rule. As indicated by the name and following the original semantics, such elements are deleted. The stereotype $\ll create \gg$ marks elements only used on the right-hand side. Clearly, such elements need to be created.

Following the mainstream syntactical style, the rewrite rule shown on Figure 1.9 models how a node labeled as “diskToMove” is disconnected from a node labeled as “lower” and is connected to a node labeled as “to”. The rule is taken from a solution to the “Towers of Hanoi” problem, discussed in [256], which is in turn based on the work of Diethelm et al. [60, 62, 61].

For this small rewrite rule, the drawbacks discussed above are negligible. However, as illustrated by Figure 1.10, the more compact rewrite syntax becomes even more important when rewrite rules are embedded within a control flow diagram. These embedded rewrite rules are called “Story Patterns”. When syntactically separating left- from right-hand sides, the two Story Patterns shown on the left of Figure 1.10 would take twice as many space.

Figure 1.9: Mainstream syntax for primitive graph rewriting rules.

Figure 1.10: Story diagram modeling a solution to the *Towers of Hanoi* problem.

Figure 1.11: Story diagrams for initializing the data structure for the *Hanoi* example.

Since this thesis' contributions (as summarized in Section 1.1.3) do not depend on the compact rule specification syntax of Story Diagrams, all examples from subsequent chapters could be mapped to the "Progres" specification style. A tool that supports such a mapping has not been constructed so far.

1.3.4.3 Control Structures

The need for control structures may be evident in a general purpose programming context. In fact, it is hard to imagine how the *Hanoi* solution shown on Figure 1.10 would be realized without conditionals or method calls. However, in a model transformation context, it should be noted that several languages that do not support conditionals, loops or invocations. Therefore, [256] motivates why such constructs are essential in a model transformation context too.

Story Diagrams offer five concepts for modeling a transformation's control flow explicitly: *sequential composition*, *conditionals*, *while loops*, *iterative loops*, and *method calls*. The following paragraphs describe these constructs in more detail.

Sequential Composition enables one to express that a story pattern *b* should be evaluated one step after another story diagram *a*, independently of whether *a*'s left-hand side could be matched. This is specified by a guard-less transition between the states of *a* and *b*.

The Story Diagram from Figure 1.11 (a) illustrates the use of an explicit conditional state.

Such a state is not related to any rewrite rule and does not have any other constraints or side-effects associated with it. However, it may put additional focus on the conditions of its outgoing transitions. The example illustrates that Story Diagrams support more condition types than the *success* and *failure* conditions introduced in the context of Figure 1.10's Story Diagram. Note that the latter two condition types only apply when a state does have a pattern associated with it.

A while loop consists of a story pattern "conditional", a *success* transition to an arbitrary sequence of "body" states, a sequential step back to "conditional", and a *failure* transition to the end of the loop. Note that the "conditional" story pattern may have side-effects. Such

side-effects are even mandatory to terminate a loop with an empty sequence of “body” states. Otherwise, the sequential transition from “conditional” to itself would be fired endlessly as soon as the left-hand side of “conditional” would match.

A frequently required task is walking iteratively over a set of pattern matches, or simply “handling all matches of a particular pattern, once”. When realizing such a traversal using a while loop, one needs to mark elements from the matches in such a manner that they will not match another time. Instead of this low-level approach, one may want to use a more explicit language concept, called “iterative loops”. Such loops visit the matches from their “conditional” pattern only once.

The final control flow construct is the method call. For object-oriented modelers, the concept of such a call should be more than familiar. Since each Story Diagram maps directly to a method, for which a Java (or C++, ...) implementation can be derived automatically, a call to a Story Diagram can be realized by means of a Java method call. Thus, the Java virtual machine’s infrastructure for stack frames is reused. Still, some may argue that a more understandable formalization of Story Diagram method calls would not rely on the infrastructure of other languages. Therefore, Zündorf illustrated how three simple concepts “H_Stack”, “H_Frame” and “H_Variable” enable one to model the behavior of a method call explicitly, using more primitive Story Diagrams [282].

Conceptually, the use of control flow structures simplifies the subgraph matching problem that is needed within a graph transformation engine, significantly. Engines no longer need to search for morphisms from the left-hand side graphs across the complete host graph. Instead, the matching algorithm builds incrementally upon results from previously matched rules. Notationally, Story Diagrams provide the so-called “bound” property to indicate what nodes are already found within the host graph. Only other nodes from the host graph need to be looked up by the matching engine.

Bound nodes come from two sources: first of all, object attributes and method parameters are available to all patterns in a Story Diagram. This is illustrated by the Story Diagrams from Figure 1.11 (a) and (b): after creating the node *left* in the first Story Diagram, it is passed to the second Story Diagram, where it is represented by the bound node called *lower*. A second source of bound nodes is the following: all nodes that have been matched or created by one rule become implicitly available to all other rules that are scheduled sequentially later, and in the same or a deeper scope. Again, Zündorf has illustrated how this implicit information can be made explicit by relying on the “H_Stack”, “H_Frame” and “H_Variable” concepts.

When supporting method calls, one requires a means to return an object or a value to the caller. In Story Diagrams, this is supported by means of end states that have the name of a particular bound node or that have a specific value as their name. When a transformation reaches such an end state, the related node or primitive value is returned to the caller.

1.3.4.4 Evaluation of the Story Driven Modeling Methodology

This subsection introduced the reader to the Story Driven Modeling process, by referring to an evaluation of the *Towers of Hanoi* solution from Diethelm et al [256].

Obviously, the Story Driven Modeling process does not provide the Silver Bullet that would enable anyone to develop applications without relying on some software engineering skills [33]. In summary, [256] illustrates that the derivation of executable behavioral models from scenario models cannot be automated completely. Consequently, the development of Story Diagrams does require abstraction capabilities from the modeler.

The good news is that the developers of model transformations are at least educated software engineers or experienced programmers. Such developers have good abstraction capabilities and may find Story Diagrams a useful formalism for modeling transformations in a human-friendly manner: on the one hand, the underlying object-oriented paradigm ensures the language concepts are familiar to the large audience of *Java*, *C#* and *C++* developers. On the other hand, pattern matching and rewriting behavior can be modeled explicitly with so-called “Story Patterns”. Moreover, several control flow constructs enable one to model the dependencies between such visual rewrite rules explicitly too.

Finally, remark that Story Diagrams consist of existing UML language constructs. Therefore, it seems promising to apply existing modeling tools for the new application domain of transformation modeling.

1.3.5 MOF Queries, Views and Transformations Standard

As discussed in Section 1.1.2, the emerging QVT standard proposes the use of (at least one of) three languages for transformation modeling:

- *QVT Relations* is a language for modeling transformations between models that need to be kept consistent in multiple directions. Change propagation is supported implicitly.
- *QVT Core* is a language for modeling the same kind of transformations. Unlike the relational language, the core language has explicit support for modeling the data structure that needs to be used for realizing traceability between related elements.
- *QVT Operational* is a language for modeling transformations from source to target models explicitly.

The relations language is supported by a graphical as well as a textual concrete syntax. The other languages have a textual concrete syntax only.

Unfortunately, the semantics of the relations and core language is not defined formalized denotationally. Therefore, it is unclear how one should realize the multidirectional constraints imposed by the expressions in these languages [240]. Interestingly, the standard does provide a mapping from the relational to the core language. However, this mapping only makes the traceability data structure of a relation more explicit. Even the semantics of the operational language is only marginally related to more well-understood concepts such as classes, methods and contracts.

Obviously, this thesis does not solve all these issues at once. However, the following techniques address particular aspects of the problems described above:

- transformations are modeled using well-understood object-oriented concepts such as classes, methods and contracts,
- method behavior is modeled in a graphical language that has been designed with cognitive modeling concerns in mind,
- it is illustrated how the semantics of new language constructs can be formalized using higher order transformations.

1.4 Summary and Outlook

This chapter motivated the need for the contributions of this thesis by describing the context of model-driven engineering and the evolution of standards and research related to transformation modeling.

In summary, several steps have been made towards a standard, object-oriented approach to the model-driven development of model transformations. Although the relevance of the OCL has been identified in the context of specific transformation case studies, the challenging nature of the problem has lead to incomplete results. Moreover, the applicability of the UML as a general purpose, extensible, transformation modeling language has been overlooked by the *MDA* initiative [192, 193, 194, 197], as well as by related initiatives such as *Software Factories* [107], *MIC* [144], and *CaMPaM* [173].

The remainder of this thesis is structured as follows: Chapter 2 makes the variety between existing languages for model transformation more concrete by means of a comprehensive taxonomy. Chapters 3 to 10 present new techniques for modeling model transformations. These chapters are divided across three parts: Part I contains transformation case studies involving transformations between models at the same level of abstraction. In contrast, output models are more detailed than input models in Part II. Part III investigates the applicability of the techniques presented so far in a model synchronization context. This part of the thesis presents more challenging open issues and directions for future work than the other two parts. Finally, the closing chapter of the thesis draws conclusions that bridge the three parts of the text.

Chapter 2

A Taxonomy of Model Transformation

This chapter is based on a paper that has been published in the electronic proceedings of a Dagstuhl workshop in 2005 [158] and two other papers that have been published in the proceedings of the GraMoT workshop in 2006 [164, 165].

The chapter presents a *taxonomy* of what has been put forward as the most profound aspect of model-driven software development [98, 234], that is: *model transformation*. By *taxonomy* we mean “A system for naming and organizing things [...] into groups which share similar qualities” (Cambridge Dictionaries Online).

Such a taxonomy can be used for a wide variety of purposes [103]. Among others, it can serve as a comprehensive overview for newcomers in the domain of interest, it can help a software developer in choosing a particular model transformation approach that is best suited for his needs, it can help tool builders to assess the strengths and weaknesses of their tools compared to other tools, and it can help scientists to identify limitations across tools or technology that need to be overcome by improving the underlying techniques and formalisms.

The need for a taxonomy has already become evident in the introductory chapter, by discussing the QVT RFP (Section 1.2.3) and the existing approaches to modeling transformations (Section 1.3): apparently competing approaches often turn out to have been developed with different design goals in mind. Since this thesis aims to be applicable to a wide range of model transformation problems, the taxonomy from this chapter introduces the reader to the different types of model transformations, the different choices leading to a particular transformation design, the different features that favor one design over another one within the context of a particular model transformation language, and finally the key features that are relevant for comparing model transformation tools. Subsequent chapters rely on the taxonomy to delimit their scope and highlight their contribution.

This chapter is organized as follows: Section 2.1 explains how the taxonomy evolved from community input. The subsequent sections cover the four taxonomy categories: Section 2.2 presents the taxonomy elements to classify *transformation types*, Section 2.3 enables one to classify *transformation designs*, Section 2.4 discusses *transformation language* features, followed by *transformation tool* features in Section 2.5. Finally, Section 2.6 summarizes the chapter and presents a brief overview of the subsequent chapters.

2.1 The Taxonomy as an organic Community Artifact

The terminology of the proposed taxonomy emerged from discussions among the participants of a 2004 Dagstuhl seminar on *Language Engineering for Model-Driven Software Development* [24, 158]. Although most discussion participants were active in the *graph transformation* and *generative programming* communities, the initial taxonomy incorporated many features from the program transformation community [273]. Therefore, we joined a new Dagstuhl working group consisting of experts in the field of *program* and *graph* transformation that were interested in comparing their techniques in the context of *model* transformation [47]. The group started with a discussion on the essential characteristics of model transformations, as well as their supporting languages and tools. The group also discussed the commonalities and variabilities between existing transformation approaches, with a special focus on bridging the program and model transformation communities.

An initial version of the taxonomy [164] has been used to compare graph transformation tools in the context of model transformation [165]. Moreover, the taxonomy was applied, and perhaps more importantly *discussed*, as a classification framework in a specialized graph transformation workshop [262]. An undergraduate course [266] also indicated that some parts of the Dagstuhl taxonomy should be emphasized more than several detailed taxonomy elements. Finally, distinguishing transformation language features from tool features has often lead to open debates, for example in the context of the AGTiVE graph transformation tool contest [214].

In parallel, Czarnecki et al. developed a model transformation survey based on similar classification concepts. Interestingly from a manageability perspective, Czarnecki's survey puts more emphasis on the hierarchical relation between concepts by relying on feature diagrams [49].

This chapter revisits the “*Dagstuhl taxonomy*” from [164] by introducing four taxonomy categories: transformation types, transformation designs, transformation languages and transformation tools. These categories should enable one to focus comparisons by discarding features that are irrelevant within a particular context. The new “transformation designs” category was introduced due to the increased maturity and convergence of transformation languages. More specifically, several languages have been extended with constructs that compensate for taxonomy features that once illustrated their limitations.

Apart from the four new categories, this chapter reorganizes the elements from [164] and [49] in order to eliminate several redundancies. Instead of following a strict hierarchical decomposition, this chapter also discusses dependencies that crosscut different categories. Finally, some concepts have been renamed or refined because their meaning turned out to be confusing due to their use in too many contexts.

2.2 Classifying Transformation Types

This section introduces the reader to well-known types of transformations in subsection 2.2.1. Subsection 2.2.2 provides a number of dimensions to distinguish between such transformation types. Without aiming to be normative, subsection 2.2.3 then illustrates how the transformation types can be classified according to these dimensions.

2.2.1 Transformation Types

This section presents commonly used names for a set of well-known model transformation types. Section 2.2.3 classifies these transformation types according to the elements from Section 2.2.2.

Synthesis involves the transformation of a higher-level, more abstract, specification (e.g., an analysis or design model) into a lower-level, more concrete, description (e.g., a model of a Java program). A typical example of synthesis is *code generation*, where the source code is translated into machine code [154], or where the design models are translated into source code [151, 1]. Another example is *formal refinement*, where a logic-based specification is transformed verifiably into an executable implementation, by means of successive refinement steps that add more concrete details [279, 11].

Reverse Engineering is the inverse of synthesis and involves the extraction of higher-level specifications from lower-level ones. A typical example is the extraction of control flow graphs from source code [250].

Migration involves the transformation from a software model written in one language or framework into another, but keeping the same level of abstraction [155].

Optimization involves improving certain operational qualities (e.g., execution time and/or memory footprint) while preserving the semantics of the software [31].

Restructuring involves changing the internal structure of software to improve certain software quality characteristics (such as understandability, modifiability, reusability, modularity, adaptability) without changing its observable behavior [108]. *Refactoring* refers to restructuring in an object-oriented context [91, 163].

Normalization involves decreasing the syntactic complexity, either by translating complex language constructs (syntactic sugar) into more primitive language constructs (*desugaring* [206]) and/or by transforming all uses of a language construct in a normal form (*simplification* [167]).

Remark that *database normalization* [19] may relate more to model restructuring according to these definitions.

Composition involves the integration of models that have been produced in isolation into a compound model. When using the term *model merging* [77], one often emphasizes the need for human interaction while the term *model weaving* is often used when a separate traceability model, holding references between corresponding model elements, needs to be produced [59].

Model Synchronization involves the integration of models that have evolved in isolation but that are subject to global consistency constraints. In contrast to model composition, changes are propagated to the models that are being integrated. When using the term *inconsistency management* [131], one may stress that not all inconsistencies need to be resolved automatically while the term *consistency maintenance* does bear the connotation of automatic enforcement of consistency constraints [187].

Model synchronization usually requires that source model changes are propagated to corresponding target model changes. This requirement is often referred to as *change propagation* [197] or *target incrementality* [49]. Unrelated – or rather: *unconstrained* – source and target elements should be left intact by the transformation engine.

Another requirement that characterizes model synchronization is the absence of a unique “source” to “target” direction. More specifically, some changes may need to be propagated in another direction than other changes. Therefore, model synchronization is a multidirectional transformation problem, involving an arbitrary amount of models where changes in one model may trigger updates in all other models.

Unlike the other transformation types, model synchronization is orthogonal to most other transformation types. More specifically, an integrated system for model synthesis and reverse engineering can be classified as a model synchronization system. Similarly, one can require that refactoring or migration tools preserve snapshots of all system variations. If the tool manages the co-evolution of these snapshots, it supports model synchronization.

A final transformation type relates to combinations of the other types: *transformation chaining* involves the combined use of reverse engineering, normalization, migration, composition and synthesis between sets of models, potentially applying model synchronization within individual transformation components [270].

2.2.2 Taxonomy Elements

This section presents three dimensions for classifying the transformation types discussed above: subsection 2.2.2.1 defines the distinction between translations and rephrasings. Subsection 2.2.2.2 defines the complementary difference between horizontal and vertical transformations. Finally, subsection 2.2.2.3 separates syntactical from semantical transformations.

2.2.2.1 Translation or Rephrasing

In order to transform models, these models need to be expressed in some modeling language (e.g., UML class diagrams for design models, and Java for source code models). Based on the language in which the source and target models of a transformation are expressed, a distinction can be made between *translation* and *rephrasing* [273]. Rephrasing involves transformations between models expressed in the same language. Such transformations are called *rephrasings*. Translation involves transformations between models expressed using different languages. Such transformations are called *translations*.

Transformation types should be classified at the conceptual level. More specifically, one should not focus on the metamodels related to particular language implementations. For example, the different UML diagram types should be considered as different languages even though they are commonly formalized by the same metamodel. Thus, a transformation that maps UML sequence diagrams to UML collaboration diagrams is classified as a translation.

2.2.2.2 Horizontal or Vertical

A *horizontal transformation* is a transformation where the source and target models reside at the same abstraction level. A *vertical transformation* is a transformation where the source and target models reside at different abstraction levels. Concrete examples are considered in Section 2.2.3.

2.2.2.3 Syntactical or Semantical

A final distinction can be made between syntactical and semantical model transformations. The former merely transform the representation of a model while the latter's output models also have a different meaning than the input models.

As an example of *syntactical transformation*, consider a parser that transforms the concrete syntax of a program (resp. model) in some programming (resp. modeling) language into an Abstract Syntax Graph (ASG), provided the input is well-formed. Syntactical transformations thus remove/add “presentation details” from/to input models. As an example of a semantical transformation, consider any transformation that manipulates that ASG for realizing a refactoring, migration or optimization.

In the context of large transformation chains, the semantics of the intermediate models tends to be changed gradually by all participating components. Therefore, according to the strict definitions, all components could be classified as semantical transformations. However, in practice, transformations with a low complexity tend to be classified as syntactical whereas more complex transformations are considered to be semantical. Therefore, it depends upon the design of a transformation chain, whether a component of a particular type (desugaring, language migration, optimization, ...) is classified as syntactical or semantical.

2.2.3 Classification

Table 2.1 illustrates that the dimensions *horizontal versus vertical* and *rephrasing versus translation* are truly orthogonal, by classifying the transformation types from Section 2.2.1 in a two-dimensional space defined by these two dimensions. As explained above, a classification in terms of all three dimensions would become application specific.

	horizontal	vertical
rephrasing	<i>API migration</i>	<i>Formal refinement, Refactoring, Optimization, Simplification</i>
translation	<i>Language migration</i>	<i>Code generation, Desugaring</i>

Table 2.1: Orthogonal classification of model transformation types.

As a clarification for the *Formal refinement* mentioned in the table, a specification in first-order predicate logic or set theory can be gradually refined such that the end result uses exactly the same language as the original specification (e.g., by adding more axioms). Since constraints become more strict over subsequent refinement steps, the models become more deterministic and therefore less abstract.

The classification of *Refactoring* as a vertical transformation type is motivated as follows: several refactorings are intended to make source code more understandable. Other refactorings make source code less understandable but more reusable. The former refactorings can be said to increase the abstraction level while the latter ones decrease the abstraction level. An argument for classifying refactorings as horizontal transformations is that abstractions remain implicit in low-level programming language patterns. Moreover, as opposed to formal refinement, the input and output models are equally deterministic.

Similarly, optimizations and simplifications are classified as vertical transformations as they make source code less human oriented and more machine oriented. Still, models re-

main equally deterministic. Therefore, one could argue that these transformations should be classified as horizontal types.

2.3 Classifying Transformation Designs

This section presents those features that are related to the design of a transformation model. These features are not directly related to the transformation types discussed in the previous section, as several valid solutions to a particular transformation problem may be designed differently according to the following features. Moreover, the following features are not bound to a particular transformation language or tool. More specifically, designs that are classified differently according to the following features may be supported within the same language.

Subsection 2.3.1 aggregates several sub-features that related to the number of input and output models, sub-models, or metamodels. Subsection 2.3.2 relates to the technical space within which a transformation is defined while subsection 2.3.3 indicates that some transformations are actually defined between such spaces. Finally, subsection 2.3.4 indicates that some transformations are defined with more use cases in mind than others. For example, some transformations are only defined for generating models from other models whereas other transformations are defined with change propagation and inconsistency management in mind.

2.3.1 Number of (Sub-)Models and Metamodels

Several features relate to the representation or storage of the input and output elements. Subsection 2.3.1.1 indicates that, at the design level, the containers of the input and output elements of a transformation not necessarily correspond to the MDA concept of a *model*. Subsection 2.3.1.2 relates to the effects of a transformation on its input elements. Subsection 2.3.1.3 indicates that input and output elements can originate from the same or a different (sub-)model. Subsection 2.3.1.4 refers to the metamodels of the input and output models. Finally, subsection 2.3.1.5 illustrates that these features are not completely unrelated to the transformation types discussed in the previous section. Moreover, the taxonomy enables one to identify the potential disadvantages of design choices that were taken implicitly.

2.3.1.1 First-class or Sub-model Transformation

The QVT specification presumes that transformations have *complete* models as inputs and outputs. This section introduces the concept of a “sub-model transformation” for enabling discussions about alternative designs.

In order to emphasize that a transformation is considering models that correspond to a complete MOF “extent” [193], one may refer explicitly to a “first-class” model. It turns out that several modeling tools store all input *and* output elements in one *integrated* first-class model. In those cases, the input and output elements that need to be integrated are divided across so-called “sub-models”. For such tools, transformations should be designed that operate on sub-models.

A sub-model conforms to the metamodel of its containing first-class model. Sub-model transformation designs require a special treatment since instances of the same metaclass are stored within the global extent of the first-class model, even when these instances are part of

Figure 2.1: Example of a sub-model transformation.

different sub-models. The taxonomy elements from the following sections apply to first-class as well as to sub-model transformations.

To make the concept of sub-models more concrete, consider Figure 2.1. The diagram visualizes the application of a sub-model transformation that translates a set of connected circles into a trapezoid and a set of connected rectangles. To avoid confusion, it should be noted that the diagram does not visualize a transformation rule. Instead, it visualizes the effect of the application of a (set of) rule(s). Also note that the example intentionally applies to an artificial (that is: *non software-engineering*) domain. This illustrates that the classification is neither UML nor MOF specific. Moreover, the domain of geometrical figures is simple enough not to distract the reader with irrelevant details. Upcoming chapters revisit this taxonomy with realistic examples of software model transformations.

The large circle within the input model represents the input sub-model. It is not a first-class model since it does not contain the elements shown on the two rows below the circle, although these elements are contained within the same model extent. From a more direct and technical perspective, the large circle is not a first-class model since it does not correspond to a model extent. Instead, the large circle is a model *element* within the extent of another first-class model.

All elements of the sub-models are connected and are assumed to have properties that enable one to collect all sub-model elements from the large circle element automatically. The trapezoid at the right side of Figure 2.1 represents the output sub-model. The example illustrates that elements representing sub-models can be of another type than the contained elements. In fact, this turns out to be the case in most practical examples.

Since the circle and trapezoid *sub-models* are part of the same model, they use *sub-sets* of the complete set of syntactical elements (i.e., the set $\{circle, rectangle, trapezoid, star\}$). However, as indicated by the dashed arrows on Figure 2.1, the sub-models still conform to the metamodel of the complete model.

Finally, remark that one can also define transformations from a sub-model to a first-class model and vice versa. Chapter 6, for example, presents a transformation from a UML sub-model to a first-class CSP model.

Figure 2.2: Examples of input-polluting and an input-destructive transformation applications.

2.3.1.2 Input-destructive, Input-polluting or Input-preserving

Input-destructive transformations destructively update or delete input elements, either to transform them into output elements, or as an implicit control flow mechanism. Input-polluting transformations add properties to input elements with the restriction that these properties should not be defined in the input metamodel. The rationale is that input-polluting transformations are assumed not to change: (i) those input properties that could be constrained by well-formedness rules, or (ii) any “semantical information” in general. Still, input pollution may be undesirable (or impossible) when no properties are allowed to (or can) remain present after the execution of the transformation. Input-preserving transformations leave all input elements intact.

As stated, a model may contain several sub-models between which input-preserving transformations are defined. This decouples the user-oriented *input preservation* feature from a more technical feature, being the design of the metamodel(s) of the (sub-)model(s) being transformed.

Figure 2.2 (a) visualizes the application of an input-polluting transformation between first-class models while Figure 2.2 (b) shows an input-destructive sub-model transformation design. Note that in the example shown on Figure 2.2 (a), we assume that “color” is not an attribute of the “Circle” metaclass. Otherwise, the transformation would have changed the semantics of the input model and would be classified as input-destructive too.

Also note that in Figure 2.2 (b), only the upper half of the nodes within the input sub-model has been destroyed. The example is designed as such to emphasize that an input-destructive design does not imply the actual deletion of *all* input elements. Instead, a transformation design is classified as input-destructive as soon as it changes the metamodel-defined content of the input model.

Note that the input preservation feature relates to the *static mode* feature that Czarnecki discusses in the context of the *DirectionKind* (*in*, *out*, or *inout*) property of QVT transformations and transformation rules [49]. However, these QVT properties are only defined in terms of complete model transformations instead of supporting the classification of sub-model transformations too.

The *UML2CSP* case study discussed in Chapter 6 is solved with an input-preserving design. Boronat and Heckel have solved the same case study with an input-destructive design

while Asztalos et al. follow an input-polluting approach [30, 8].

2.3.1.3 In-place or Out-place

Transformations can either be executed within one (sub-)model or between two (sub-)models. The former kind of transformations is classified as *in-place* while the latter kind is known as *out-place* transformations.

Models that are manipulated by an in-place transformation evolve gradually from the input role to the output role. Therefore, in-place transformations are automatically input-destructive. Out-place transformations, however, can be realized according to the other two designs as well. Thus, an input-destructive design is not necessarily in-place. Figure 2.2 (b) for example visualizes the application of an input-destructive, out-place transformation.

This classification can also be applied at the fine-grained level of transformation rules. A primitive rewrite rule that creates or updates elements without modifying matched elements can easily be classified as out-place. Rules that rely on parameters or other bound nodes can only be classified if such nodes are known to be input or output elements.

A transformation can be classified as an in-place model transformation and an out-place sub-model transformation at the same time. In fact, such a model transformation can be classified further as input-destructive while from the viewpoint of the sub-model transformation it could be either input-polluting or input-preserving. The relative nature of this classification can be made more concrete by revisiting Figure 2.1.

The Figure shows an out-place, input-preserving sub-model transformation that is also an in-place, and thus input-destructive transformation of a first-class model. The first-class transformation is input-destructive because the state of the input model is clearly different after the application of the transformation: not only have new elements been *added*, existing elements have been *updated* too. More specifically, after the transformation, the input model also contains the *new elements* from the output sub-model. Moreover, the sets of links attached to the *existing* large circle and the central star have been *updated*.

2.3.1.4 Endogenous or Exogenous

In-place transformations are executed within one sub-model, or in extremis in one complete model. Since models statically conform to only one metamodel, one could presume that all in-place transformations are defined in terms of only one metamodel. Such transformations are called *endogenous*.

However, when using a dynamically typed language such as Smalltalk [89], a transformation definition can retype input model elements dynamically from source metaclasses to target metaclasses. In that case, the transformation does have an output metamodel that differs from its input metamodel. Such a transformation is called *exogenous*. Out-place transformations can obviously be endogenous or exogenous too.

In general, endogenous transformations are defined between models that all conform to the same metamodel. As soon as different metamodels are involved, the transformation is called *exogenous*.

Intuitively, one may presume that rephrasings are always realized using an endogenous design. This stems from the fact that rephrasing involves the transformation of models in the same language. From this fact, it may seem obvious to use the same metamodel for representing the input and the output models. However, when realizing an API migration, one

may want to rely on two different metamodels. This enables one to represent the concepts from the source and target API explicitly. Thus, some rephrasings do require an exogenous transformation design.

2.3.1.5 Relation to Transformation Types

This section discusses whether the design choices discussed above are compatible or in conflict with the transformation types from Section 2.2.1.

In-place transformations cannot be used for model synchronization since by definition, such a transformation destroys its input (sub-)model. Therefore, persistent relations between input and output elements cannot be defined. More generally, since source and target models never co-exist, one cannot propagate changes between them.

When the elements of an output (sub-)model are heavily dependent on input elements, an out-place transformation design requires a large amount of copy instructions. Although the program transformation community has developed optimization techniques for implicitly sharing nodes instead of creating actual copies, these techniques have not yet been applied in a model synchronization context.

Rephrasings are commonly realized using an input-destructive design. In fact, I am unaware of concrete solutions that rely on explicit copies of software models in the context of refactoring, simplification or optimization transformations. When such copies are not needed, an input-place design is more appropriate than an out-place design since the former requires less transformation rule applications. More specifically, an in-place approach automatically preserves all elements that are allowed to be unaffected by the rephrasing. In contrast, an out-place design requires the explicit transfer of such elements. Although out-place based frameworks, such as ATL, can provide default copying behavior for simulating in-place behavior [126], such a design is rather inefficient and is complicated by identity issues.

Translations tend to be realized using an out-place design. More specifically, translations are commonly realized as exogenous transformations on first-class models.

However, as illustrated by the examples from Figures 2.1 and 2.2, other designs are possible. In these examples, the languages involved are formalized by one integrated metamodel, thus enabling an endogenous transformations design.

A translation from UML sequence diagrams to collaboration diagrams for example can rely on the integrated UML metamodel. In that case, an out-place, input-preserving sub-model transformation design can be realized by storing the collaboration diagrams in a package hierarchy parallel to that of the sequence diagrams. These package elements play a role that is equivalent to that of the large circle and trapezoid elements in the example from Figure 2.1.

In practice, a minimal form of input-destruction can be tolerated: one may for example support traceability by creating UML dependency links between the elements from the sequence and collaboration diagrams. On the other hand, the out-place sub-model design can be kept fully input-preserving by relying on an external traceability mechanism. The transformations from Part II and III are designed as such.

2.3.2 Technical Space

A technical space [23] is a model management framework containing concepts, tools, mechanisms, techniques, languages and formalisms associated to a particular technology. A technical space is determined by the meta-metamodel that is used.

For example, the world-wide web consortium (W3C) promotes the *XML technical space*, which uses *XML Schema* as meta-metamodel. This space includes support for languages such as HTML, XML, XMI, XSLT, and XQuery. As another example, the OMG promotes the *MDA technical space*, which uses the *MOF* as meta-metamodel, and supports languages such as UML. Many other technical spaces are available, including those relying on abstract syntax trees and grammars, database technology, or ontologies.

Given a model transformation, its source and target models may belong to the same or to different technical spaces. In the latter case, we need tools and techniques to define transformations that bridge technical spaces. One possibility is to provide model exporters and importers while executing the actual transformation in the technical space of either the source or target model.

For example, when translating XML documents into UML diagrams one can choose to execute the actual transformation in either the XML or the MDA technical space. To perform the transformation in the XML technical space, one would use an XSLT or XQuery program translating the general XML document into an XML document conforming to the syntax of the XMI standard (XML metadata interchange) and conforming to the semantics expressed by the XMI representation of the UML metamodel. An XMI parser can then be used to import the resulting XMI document in a UML CASE tool, residing in the MDA technical space.

Performing the transformation in the MDA technical space would require a MOF meta-model for XML. After parsing the XML document into instances of this metamodel, the actual transformation could be performed as a MOF transformation. Similar to the QVT initiative, this thesis aims to provide a standard modeling language for transformations in the MDA technical space.

2.3.3 Intra- or Inter-Space

Transformations whose output models can be processed in another technical space are called *extractors* while transformations that can read input models from another technical space are called *injectors* [22]. In practical discussions [128], the concepts may be defined in terms of “loading” (resp. “saving”) models to (resp. “from”) the one specific tool, which is then assumed to correspond to “the” MDE space:

“An injection (resp. extraction) is a transformation from (resp. to) another Technical Space (e.g., XML, Grammarware) to (resp. from) the Model Engineering Technical Space. An injector (resp. extractor) is a tool implementing an injection (resp. extraction).” [128]

More generally, one should be able to implement injectors and extractors within any space. Coming back to the XML to UML transformation example discussed above, the XQuery program translating a general XML document to an XMI conform document can be described as an XML space extractor to the MDA space. In this context, an implicit assumption may be that injectors and extractors should only handle syntactical transformation concerns.

With this assumption, suppose an XQuery program mixes the actual mapping from XML schema semantics to UML metamodel semantics with the mapping to XMI syntax. The injection/extraction concepts now assist a design discussion. For example, one may argue to refactor all extraction code into a separate XQuery function called “extractToMDA”.

Another generalization of the concepts of injection and extraction relates to the focus on “loading” and “saving”. These terms undesirably imply that external data is read (resp.

written) to (resp. from) a fully materialized data structure. An obvious alternative is to provide in-memory adapters that avoid redundant memory usage and copy operations that may involve problems with model element identities. Jakob even facilitates the implementation of such adapters by generating the superclasses/interfaces that define the integration interface [123].

2.3.4 Versatility: Enforceable, Checkable, Ignorable, Prioritizable

The versatility of a model transformation design determines whether the transformation can only be executed to *enforce* an *implicit* consistency constraint, whether one can also *check* such a constraint *explicitly*, or whether one can both *check* for consistency *and enforce* it when it is violated. More powerful (but potentially more complex) transformation designs even enable one to tolerate particular inconsistencies, thus disabling the automatic execution of a transformation for a particular constraint violation. Finally, some designs enable one to sort, and thus prioritize detected inconsistencies. When comparing the complexity of transformation implementations, one should take into account the versatility of the designs to enable a fair comparison. Model synchronization requires at least a *checkable* transformation design. Remark that some languages, like QVT *Relational* and *Core*, favor checkable designs while other languages, such as QVT *Operational*, are not designed with checkability in mind.

2.4 Classifying Transformation Languages

This section presents an overview of features for classifying transformation languages. Subsection 2.4.1 demystifies the overloaded concept of a declarative transformation language by precisely defining the concept in terms of more concrete language features. Subsection 2.4.2 aggregates existing work for assessing the extent in which a particular language has been designed with those factors in mind that are known to affect the understandability of models. Subsection 2.4.3 presents features related to the decomposition of transformation models. Subsection 2.4.4 refers to features related to genericity and subsection 2.4.5 discusses transformation language support for traceability.

2.4.1 Declarative or Operational

A major source of speech confusion and misunderstandings is the overloaded semantics of the adjective “declarative”. Instead of pointing to all possible meanings of the adjective in general, this section points to its special meaning in a model transformation context. This model transformation language taxonomy starts from the QVT RFP, since this was a standardization initiative for model transformation languages. The RFP states [194]:

“The transformation definition language shall be declarative in order to support transformation execution with the following characteristic: Incremental changes in a source model may be transformed into changes in a target model immediately.”

Interestingly, the RFP refers to a very specific characteristic that may be provided by adding a dedicated construct to languages that may not have been classified as declarative languages according to general definitions such as [93]:

“Any relational language or functional language. [...] Declarative languages contrast with imperative languages which specify explicit manipulation of the computer’s internal state; or procedural languages which specify an explicit sequence of steps to follow. The

most common examples of declarative languages are logic programming languages such as Prolog and functional languages like Haskell.”

Even more interestingly, the information whether each output model is created anew or whether changes can be applied to existing models incrementally too is not automatically left implicit in Prolog or Haskell implementations of model transformations.

Therefore, this taxonomy decomposes the definition of declarativeness in terms of three other features. More specifically, a transformation language is said to be declarative “in terms of” these features if it enables one to model (i) change propagation, (ii) execution direction and (iii) rule scheduling in an implicit manner.

Finally, this section clarifies the difference between constructive and restrictive languages, since that classification turns out to be orthogonal to the presented definition of declarative while outside the model transformation context, restrictive languages have often been called declarative too.

2.4.1.1 Change Propagation: Implicit, Explicit or Both

Transformations can check whether traceability links are connected to input model elements. If such links are present, the input model elements have been mapped to output elements before. Instead of relying on traceability links, a transformation approach may require one to define keys on the metaclasses of mapped elements. This may enable a transformation engine to look up existing elements before creating new ones in the target model. As long as such elements satisfy the constraints enforced by the transformation, from the viewpoint of the key properties, there is no need to replace them by new elements.

It should be noted that change propagation not only involves the addition of new elements or the updating of existing ones. In some cases, target elements may need to be deleted too. If changes can be propagated even when all change and update scenario’s are not modeled explicitly, the transformation language offers implicit change propagation. In the case of explicit change propagation, one transformation rule or branch within a large transformation rule needs to be provided for every change scenario.

Note that some transformation languages may offer language constructs for implicit change propagation while also offering operators for modeling the resolution of concrete inconsistencies explicitly. The latter operators may be used when the default change propagation behavior is different from the expected behavior in a concrete case.

2.4.1.2 Execution Direction: Implicit, Explicit or Both

When models need to be synchronized in multiple directions, one may want to model the transformation using language constructs with an implicit execution direction. This results in transformation models that are more compact than when only using language constructs that operate in one execution direction explicitly.

Remark that a particular language may have a declarative semantics in theory whereas its concrete syntax may favor an operational default implementation. Such languages may lead to abusive reliance on the behavior of a particular execution engine. More tragically in practice, such languages may lead a debugger to wrong assumptions about the semantics of a particular transformation model. To clarify this rather subtle issue, consider the following fragments of transformation code, slightly adapted from examples in the QVT specification [197]:

```

1 top relation PackageToSchema {
2   pn: String;
3   enforce domain uml p:Package {
4     name=pn
5   };
6   enforce domain rdbms s:Schema {
7     name=pn
8   };
9 }

```

```

1 map packageToSchema in umlRdbms {
2   uml () {
3     p:Package
4   }
5   rdbms () {
6     s:Schema
7   }
8   where () {
9     p2s:PackageToSchema |
10    p2s.schema = s;
11    p2s.umlPackage = p;
12  }
13  map {
14    where () {
15      p2s.name := s.name;
16      p2s.name := p.name;
17      s.name := p2s.name;
18      p.name := p2s.name;
19    }
20  }
21 }

```

The example transformation definition on the left is written in the QVT *Relations* language. This example expresses that a *Package* element should always be mapped to a *Schema* element with the same name. The transformation is modeled declaratively in terms of the direction in which potential inconsistencies should be resolved. More specifically, when a *Package* element is related to a *Schema* element with a different name, the transformation engine can update the name of either of these elements.

Once the left fragment is understood, consider the fragment on the right. That example is written in the QVT *Core* language but models the same kind of transformation behavior than the previous one. More specifically, since the order for enforcing the constraints from line 15 to 18 is undefined, the inconsistency described above can still be resolved in either direction. However, a default implementation of QVT Core may always enforce these constraints in the order of their definition.

In this concrete example, the constraint on line 16 would be enforced after that on line 15. Consequently, the link element, named *p2s*, would always have the same name as the *Package* element at the point where the constraint on line 17 would be evaluated. Thus, when a *Package* element *p* would be related to a *Schema* element *s* with a different name, the transformation engine would always overwrite the name of *s* with that of *p*. Transformation writers may start relying on that behavior which may lead to unexpected results on other compliant engines. This may hold especially for developers with an imperative programming language background that are “suddenly” supposed to maintain this kind of transformation specifications.

Note that the symmetrical modeling style imposed by the QVT *Relations* language (cfr., the fragment on the left) may lead less transformation writers to wrong assumptions concerning the implied execution direction.

In summary, one should use this taxonomy with care: although it is convenient, and therefore tempting, to ignore subtleties such as the issue described above, two languages that are both labeled as “fully declarative” may still impose a different mindset on the transformation modelers.

Figure 2.3: Transformation Language features for Rule Scheduling

2.4.1.3 Rule Scheduling: Implicit, Explicit or Both

Figure 2.3 visualizes the different features related to rule scheduling as a feature diagram [51]. Although the feature diagram syntax is also applied in the classification of Czarnecki [49], the diagram from Figure 2.3 is based on our new classification. While similar diagrams could be constructed for other parts of this taxonomy, we believe the visualization is particularly relevant for rule scheduling, due to the multitude of related subfeatures.

A transformation rule is a unit that maps elements from the input model(s) to elements from the output model(s). Transformation rules need to traverse the source and target models(s). Source model navigation is based on pattern matching within graphs of model elements and their relationships. The mechanics of the pattern matching can be implemented explicitly in a general purpose programming language or it can be left implicit by dedicated language constructs. In this context, Section 1.3.4.1 discussed the role of the left-hand side of primitive graph transformation rules. Some languages have rather unique pattern specification constructs. *AGG*, for example, supports Negative Application Conditions (NACs [110]). As another example, *GROOVE* supports regular expressions on the edge labels within graph patterns [212].

Some transformation languages have implicit rule selection algorithms, where some non-determinism makes abstraction from the exact nature of the traversal. Other transformation languages have explicit constructs for controlling the flow of transformation rules, and some transformation languages support both modeling styles. Apart from the commonly known explicit control flow concepts (loops, structured conditionals, ...), the feature diagram shown on Figure 2.3 contains several features for classifying perhaps lesser known languages supporting implicit rule selection. The most implicit kind of rule selection is *unconstrained*: there are no hierarchical dependencies or priorities between rules. As soon as the transformation is started, and each time a rule has been executed, all rules are evaluated.

This taxonomy presents three mechanisms to constrain rule selection: either *priorities* between rules indicate that one set of rules should be evaluated before another set of rules. The scheduling of rules with the same priority remains unconstrained. *Layers* are rule sets that are scheduled according to the kind of side-effects they perform. Finally, *deferred* execution can be considered as a very primitive layering mechanism that enables one to specify that a fragment of a rule should be executed after the side-effects of all other rules have been performed.

*AToM*³ supports rule priorities, while *AGG* supports rule layers based on labeling functions that associate model element properties with a layer number [79, 247]. Rule layers can be constrained such that, for example, each rule in a layer only deletes those elements that are

labeled with the number of a lower layer. Finally, deferring the execution of rule fragments is supported by *QVT Operational*.

Since a rule may match simultaneously on several locations within the input model, rule scheduling can be classified further based on the scope of *rule application*. Either the rule is applied atomically on all its matches, or it is applied on only one match in the current iteration. The former application style is often called “concurrent” application while Czarnecki refers to the latter style as “one-point” application. Both application styles are supported by *AToM³*.

Automatic rule scheduling can be complemented by *interaction* with the modeler, both at the level of implicit rule selection and at the level of rule application. Interaction at the rule selection level concerns the manual selection of one rule among a set of matching rules. This can occur in the case of unrestricted rule selection as well as in the case of constrained rule selection. In the case of priorities, for example, it is undefined which of two rules with the same priority should be selected when matches for both rules are found. Interaction at the rule application level concerns involves the manual selection of one particular match when a rule is selected that matches on multiple input model locations. In *AToM³*, interaction is supported at the rule application level.

It may be noted that the first two features related to the “declarativeness” of a transformation language have more impact on the resulting type of transformation than the latter feature does. More specifically, when applying language constructs with implicit change propagation and an implicit execution direction, the resulting transformation can automatically be used for model synchronization. Using a language with implicit rule scheduling does not automatically result in a comparable increase in versatility. However, when using layers for constraining the rule selection, one may automatically ensure some properties like termination [74].

2.4.1.4 Restrictive or Constructive

Some languages, or well-defined fragments thereof, only enable one to constrain data. Such *restrictive* languages can be used to define constraints between the input and output model elements of a transformation rule. Other languages can be used to construct or update data explicitly. Such *constructive* languages can be used to create output model elements based on input model elements [10].

In the context of automating (at least a part of) the application of model transformations, the use of restrictive languages is motivated as follows:

- restrictive languages may be analyzed more easily than constructive languages,
- for some models in a restrictive language, one can derive constructive behavior automatically.

Coming back to the languages that have been proposed for modeling transformations in a standard-compliant manner (see Section 1.3 from the introductory chapter), one can classify the OCL, QVT Relations and QVT Core as restrictive whereas Story Diagrams and QVT Operational can be classified as constructive.

In this thesis, Chapter 3 discusses the use of OCL in a model refactoring context, while Chapter 9 presents the combined use of OCL (a restrictive language) and Story Diagrams (a constructive language). Note that constructive languages can be used for modeling transformations restrictively too but the compilers or interpreters for such languages do not tend to support the automatic derivation of constructive behavior from such models. However, as

Chapter 10 motivates, it may be useful to go beyond that tradition and investigate the strengths and drawbacks of “hybrid transformation languages” in more detail.

2.4.2 Concrete Syntax

Obviously, a transformation language can be either visual or textual. Since the main purpose of this taxonomy is to enable one to classify approaches, it will not argue which approach is most appropriate for modeling transformations. In fact, the superiority of one approach is too often assumed, or even claimed, without any link to empirical evaluations [105, 277]. In practice however, the fitness of a language design may heavily depend on one’s experience [182] or preference [41] and may be influenced significantly by training [205].

Therefore, the visual programming community is consolidating positive and negative results from empirical studies in a framework of cognitive dimensions [104, 27]. The use of the framework does not guarantee that one’s language will be accepted by its users, however it does provide a useful set of considerations that emerged from more than two decades of empirical research in the domain. Moreover, as Green and Petre [106] state, “with a defined vocabulary in place it becomes much easier to describe how remedies for weaknesses can be provided, [...]”. In that context, the framework of cognitive dimensions, quoted from [106], fits directly in this taxonomy for model transformation languages:

Abstraction gradient: What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?

Closeness of mapping: What “programming games” need to be learned?

Consistency: When some of the language has been learnt, how much of the rest can be inferred?

Diffuseness: How many symbols or graphic entities are required to express a meaning?

Error-proneness: Does the design of the notation induce careless mistakes?

Hard mental operations: Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what’s happening?

Hidden dependencies: Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?

Premature commitment: Do programmers have to make decisions before they have the information they need?

Progressive evaluation: Can a partially-complete program be executed to obtain feedback on “How am I doing”?

Role-expressiveness: Can the reader see how each component of a program relates to the whole?

Secondary notation: Can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the official semantics of the language?

Viscosity: How much effort is required to perform a single change?

Visibility: Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

2.4.3 Decomposition Mechanisms

Like any other programming language, model transformation languages need decomposition and modularization mechanisms. Most languages support *modules* as a means to structure a large set of transformations. In general, *transformations* modularize rules. In some languages, such as QVT Operational, rules can be embedded in the definition of another rule.

Czarnecki and Helsen discuss the different rule decomposition hierarchies that can be enforced by the design of a transformation language [49]: the rule hierarchy can either mimic the source or target language, or it can have an independent structure. The approach presented in this thesis supports any decomposition structure.

Lenyel et al. describe why and how constraints that are tangled over multiple rewrite rules should be in an *aspect oriented* manner [147]. In this thesis, Chapter 6 illustrates how a complex transformation rule can be decomposed in multiple, smaller and overlapping, *transformation views*.

2.4.4 Genericity

This section presents three features related to genericity: *specialization* relates to reusability, *overloading* relates to calling an implicit dispatch method, *reflection* relates to the implicit use of the source/target metamodels, and *higher order transformations* relate to the extensibility of the transformation language.

2.4.4.1 Specialization

Specialization at the transformation level corresponds to the inheritance of a set of rules. At the rule level, the semantics of specialization depends on whether implicit or explicit rule scheduling is employed:

- For explicitly schedulable rules, specialization corresponds to method overriding.
- For implicitly scheduled rules, the specializing rule can define additional restrictions for the specialized rule. Moreover, all side-effects of the specialized rule are still applied by the evaluation engine, even when the specializing rule defines its own side-effects.

The transformations from Part II rely on method overriding at the implementation level. The *QVT Relations* language supports the specialization of implicitly scheduled rules [197].

2.4.4.2 Overloading

Overloading refers to the ability to define several operations (i.e., explicitly scheduled rules) with the same name, yet with different parameter lists. An overloaded operation call is dispatched rather implicitly, based on the types of the “message” arguments. In contrast, a conventional call can be dispatched by only looking at the message name.

Remark that only languages that support “multiple dispatch” (also known as “multi-dispatch”, “dynamic dispatch”, “multiple polymorphism”, or “multi-methods”) enable one to call an overloaded method in a generic manner safely [12, 67, 280, 120, 88]. The refinement from Chapter 6 relies on overloading to model all transformation rules of that transformation in a uniform manner. The chapter discusses how genericity has been realized on an underlying platform that only supports single-dispatch.

2.4.4.3 Reflection

Reflective model transformation languages enable one to treat metamodels as first-class models. Moreover, such languages enable one to navigate from a model element to its metaclass. On the one hand, this metaclass can be treated as a regular model element. This enables one to query for all properties of a particular metaclass. On the other hand, the metaclass can be used to constrain other model elements. More specifically, it can be used to test whether or not another model element is of that type too.

Chapter 8 illustrates this kind of reflection. Varró and Pataricza define the support for reflection in *VIATRA2* as follows [272]: “Generic transformations contain transformation rules where the types of certain objects are variables.”

2.4.4.4 Higher Order Transformations

Transformation languages with support for higher order transformations enable one to treat transformations as first-class models. This enables one to use transformation models as input and/or output of other transformations.

Chapters 7 and 8 rely on a higher order transformation to normalize a new transformation language construct into more primitive language constructs. Varró and Pataricza refer to higher order transformations as meta-transformations [272].

2.4.5 Traceability

Traceability has already been discussed in the context of change propagation in Section 2.4.1.1. That discussion indicated that traceability links can be created, navigated and updated either implicitly or explicitly. In the case of *explicit link manipulation*, one can distinguish further between *implicitly defined* links and *explicitly defined* links.

Languages supporting implicitly defined links may not provide constructs to manipulate or access these links explicitly. *QVT Relational* is an example of such a language. In this case, links even do not have to be created. However, as illustrated by the mapping from *QVT Relational* to *QVT Core*, such links may be used behind the scenes.

QVT Core is an example of a language where links have to be *defined* and *constrained* explicitly (using MOF and OCL) while these links are *manipulated* (created, traversed, updated and deleted) implicitly.

The QVT standard does not provide a language to express explicitly how the links and input/output model elements are manipulated to maintain the consistency rules expressed by the OCL expressions in QVT predicates. This thesis thus complements the standard by illustrating how such behavior can be modeled explicitly in Story Diagrams. Moreover, Part III illustrates how such Story Diagrams can be derived automatically from Triple Graph Grammar rules that are at the same level of abstraction as QVT relations.

2.5 Classifying Transformation Tools

This section presents a set of features to classify transformation tools according to features that are not bound to the transformation language. Subsection 2.5.1 indicates that not all modeling tools enable the deployment of new transformations or the modification of existing ones. Subsection 2.5.2 clarifies the distinction between practical usability issues and more fundamental concerns about the usefulness of a tool. Subsection 2.5.3 puts the performance of a tool in perspective. Subsection 2.5.4 discusses the different mechanisms for accessing model elements while subsection 2.5.5 discusses the role of standards in a model transformation context. The techniques that are contributed by the upcoming chapters are supported by a tool prototype called MoTMoT. To illustrate it was designed with this taxonomy in mind, the subsequent subsections point to particular features of that tool.

2.5.1 CRUD: Creating/Reading/Updating/Deleting transformations

While this is a trivial requirement for a transformation language, it is not that obvious for a transformation tool to support the creation of new transformations, or the updating or deletion of existing ones.

For example, if we consider a typical program refactoring tool, it comes with a predefined set of refactoring transformations that can be applied, but there is often no way to define new refactoring transformations, or to fine-tune existing transformations to specific needs of the user. As such, having the possibility to create and deploy new transformations or update existing ones is an important criterion.

2.5.2 Useful and Usable

The constructs of a tool's transformation language should be *useful*, which means that it has to serve a practical purpose. The taxonomy elements from the previous section enable one to make a project-specific assessment of useful versus irrelevant language features. On the other hand, the concrete tool needs to be *usable* too, which means that it should be intuitive and efficient to use.

Obviously, this issue is partly related to developer training and experience. However, some tools are completed to the point where they are sufficiently usable for industrial applications. MoTMoT [176], for example, was primarily built to validate the techniques from this thesis. Since it does not provide an interactive semantical analyzer or debugger, it is not usable within large-scale projects.

On the other hand, the Fujaba toolsuite illustrates that such functionality can be built for Story Diagrams and OCL [146, 242]. Therefore, it should be stressed that many (though obviously not all) usability concerns are only related to the tool. When one is convinced that a language is useful enough for industrial application, one should invest sufficiently in the implementation, or acquisition, of a tool for that language.

Finally, one should remark that this thesis is not limited to increasing the usefulness of a particular model transformation language. Instead, Chapter 5 presents how MoTMoT's design enables more reuse of tool components. In fact, when using MoTMoT, one benefits from new usability features that were developed by maintainers of general UML diagram and profile components. In an industrial context, this would free the maintainers of a Story Diagram

plug-in from building these features as well. Thus, they could invest their effort in other usability oriented features.

2.5.3 Run-Time Performance versus Developer Performance

The language or tool should be able to cope with large and complex transformations or transformations of large and complex software models without sacrificing performance.

One feature that is often associated with performance is whether the transformation tool is interpreter-based or compiler-based. Compilation-based approaches may have the benefit of improved performance over interpretation-based ones.

On the other hand, compilation-based approaches may require a complex and time-consuming “build, compile, test, deploy” cycle. Dynamic environments may therefore result in better developer performance. In practice, the choice between a compilation and interpretation may also depend on the target (legacy) platform, as illustrated by the following section. At the lowest level, MoTMoT relies on compilation. However, Chapter 6 (i.e., Section 6.3.5.3 in specific) illustrates how interpreter techniques can be applied at the appropriate level of abstraction too.

2.5.4 Input/Output: In-Memory versus Serialized

Transformation tools can manipulate input and output models within a modeling tool directly, using the API of that tool’s repository. Alternatively, models can be imported and exported to a string representation. MoTMoT supports in-memory and serialized input/output processing.

2.5.5 Standards

Model transformations can be standardized at two levels: first of all, the exchange of transformation definitions is enabled by using a standard transformation language. For example, one will be able to exchange QVT conform transformation models when competing tools become available. One may also rely on the UML based transformation modeling approach presented in this thesis. Secondly, one can standardize the execution level. For in-memory execution, this requires the use of a standard such as JMI whereas standard execution on serialized inputs and outputs is supported by standards such as XMI. MoTMoT supports both standards.

2.6 Summary and Outlook

This chapter presented a taxonomy of model transformation that emerged from various working group discussions, from a literature study and from performing the case studies supporting the remainder of this thesis. Unlike previous versions of this taxonomy and the prominent survey from Czarnecki and Helsén, this chapter clearly separates transformation language features from features that relate to other transformation artifacts.

More specifically, the taxonomy consists of four categories: features from Section 2.2 enable one to describe the difference between refactoring, synthesis, and other transformation types: such types take care of syntactical or semantical differences by rephrasing or translating in a horizontal or vertical direction. Section 2.3 can be used as a design checklist: does a particular solution support the design requirements enforced by your environment? Section 2.4 enables one to classify a complex set of transformation languages, as illustrated by a “dissection” of the three languages from the QVT standard. Finally, Section 2.5 closes the taxonomy with transformation tool features. Figure 2.4 visualizes the core taxonomy elements.

The main contributions of this chapter are as follows: first of all, its novel classification of transformation design features enables one to reason about technical aspects of a transformation without going into the specifics of a particular language. Remarkably, these design features are highly orthogonal. Several examples illustrate combinations that occur in practice, while they may have seemed unreasonable at first sight. Secondly, the taxonomy demystifies the concept of a “declarative” transformation language by defining it in terms of three language features. Finally, the taxonomy uses a combination of hierarchy and contains cross-references to decompose the complex set of features into a manageable structure.

This taxonomy provides a promising skeleton for classifying existing solutions (according to the features for transformation types and transformation designs) and existing transformation languages and tools. Although this chapter already referred to some interesting examples, it did not aim to provide a complete survey of the field. An extension of this chapter will be constructed for this purpose. In fact, it may be promising to maintain a taxonomy-driven survey on the Internet, since that may stimulate cross-fertilization across technical spaces.

In a nutshell, the transformations from the subsequent chapters can be classified as follows: Part I presents in-place (hence input-destructive) transformation designs of refactorings. In Part II, Chapter 6 presents a transformation chain that consists of an in-place normalization and an out-place synthesis transformation. Chapter 7 presents an out-place, input-preserving design of sub-model synthesis. Chapter 8 presents an in-place design of first-class model desugaring. This chapter concerns a second order transformation. Finally, Part III presents in-place designs of sub-model synchronization in the context of synthesis.

Figure 2.4: Concept map for the domain of Model Transformation. The dotted edges illustrate how design, language and tool features can support a particular transformation type.

Part I

Refactoring

Chapter 3

Towards Automating Source-Consistent UML Refactoring

This chapter is based on a paper that has been published in the proceedings of the UML conference in 2003 [268]. The chapter evaluates the use of OCL contracts in the context of refactorings. More specifically, OCL is used to describe the preconditions, postconditions and code smells related to two refactoring operations that transform elements at different levels of granularity. The OCL constraints are expressed on a minimal extension of the UML metamodel. This enables one to reason about source-code consistency without relying on language-specific language constructs. The chapter concludes by presenting the strengths and weaknesses of OCL refactoring contracts.

The Unified Modeling Language can be used for making refactoring tools less language specific in several ways. First of all, a UML based refactoring tool enables software designers to restructure applications without worrying about the syntaxes of the various programming languages in use. Secondly, the UML metamodel can be used as a structural model of the repository holding the elements that are common across the programming languages supported by the refactoring tool. Thirdly, the OCL can be used to specify transformation contracts that would otherwise be implemented in the programming language of the refactoring tool. Such transformation contracts enable one to compose primitive refactorings, verify preservation of program behavior, and trigger refactorings based on code smells. An MDA tool implementing these concepts allows one to improve existing UML designs while keeping them synchronized with the underlying code base.

This chapter focuses on the third aspect, being the use of OCL for specifying refactoring contracts. The fitness of the UML metamodel for realizing language-independent refactorings is outside the scope of this thesis but has been investigated in [268]. Subsequent chapters will illustrate that Story Diagrams, a UML based graph transformation language, can be used to implement the actual refactoring transformations. Thus, the UML provides sufficient formalisms to build a powerful refactoring engine for multiple languages.

The remainder of this chapter is structured as follows: Section 3.1 introduces the reader to the field of refactoring and the related tools. Moreover, the section presents the challenges

that need to be tackled when refactoring in the context of UML tools. In section 3.2 we cover two sample refactorings to illustrate how refactoring pre- and postconditions can be specified in OCL. This enables the runtime verification of behavior preservation of a refactoring engine. Section 3.3 illustrates two applications of automated UML refactorings, namely composition of refactorings, and detection of code smells. After giving an overview of some related work on refactoring in general and in a UML context specifically, the chapter presents open issues in Section 3.4 and comes to a conclusion (Section 3.5).

3.1 Refactoring: Context, Tools, Relation to the UML

An intrinsic property of software in a real-world environment is its need to evolve. As the software is enhanced, modified and adapted to new requirements, the code becomes more and more complex and the original design slowly erodes. Therefore, it is not surprising that the major part of the total software development cost is devoted to software maintenance [148, 109, 188, 112]. What may seem surprising though, is that better software development methods and tools do not reduce but *enlarge* the maintenance cost [101]. This is explained by the observation that better methods and tools are mainly used to accommodate even more requirements, thereby increasing the rate of change and consequently the effects of erosion.

To cope with design erosion there is a need for techniques that reduce software complexity by incrementally improving the internal software structure. The research domain that addresses this problem is referred to as *restructuring* [7, 108]. In the domain of object-oriented software development however, the term *refactoring* is used instead, where it is defined as “behavior preserving program transformation” [202, 91]. Refactorings are based on the redistribution of classes, variables and methods across the class hierarchy in order to facilitate future adaptations and extensions. Especially with the recent trend in agile software development, refactoring receives widespread attention and consequently many integrated development environments (IDEs) are incorporating refactoring features into their tools [276, 152].

3.2 Describing UML Refactorings as Refactoring Contracts

This section illustrates that the UML metamodel extensions from [268] are sufficient to reason about refactorings for common OO languages. Subsection 3.2.1 introduces the concept of a *refactoring contract*, a set of constraints that describe the effect of a refactoring. Next, we use OCL to describe the refactoring contracts of 2 sample refactorings of realistic complexity: subsection 3.2.2 models the *Extract Method* refactoring whereas subsection 3.2.3 relates to the *Pull Up Method* refactoring. Most OCL constraints use user-defined properties for the sake of readability. Some of the queries defining these properties are presented to give an impression of the mechanism. Subsection 3.2.4 concludes this section with a brief discussion.

3.2.1 Refactoring Contracts

A refactoring contract consists of three sets of constraints per refactoring:

1. a *precondition* describes the model restrictions that need to be satisfied before applying the refactoring,

2. a *postcondition* states what model properties are guaranteed by the refactoring, and
3. the *code smells* describe the problematic model constructs that can be improved by the refactoring.

By carefully selecting the pre- and postconditions of a contract, certain behavior properties can be guaranteed. The expressions presented in this thesis are based on the preservation of *access*-, *update*- and *call-behavior* [160]. However, no formal proofs about these properties are presented since this thesis only aims to evaluate the relevance of using OCL in this context. This relevance is investigated by analyzing which properties of the language are considered useful on the one hand and which drawbacks and limitations require complementary techniques on the other hand.

3.2.2 Extract Method

The “Extract Method” refactoring turns a sequence of statements from a method *oldM* in class *C* into a new method *m* in *C*. The aim of this refactoring is to remove code that is duplicated across multiple methods of the same class and / or increase the intentionality of a *long method* [91].

3.2.2.1 OCL Pre- and Postconditions

The following OCL fragments support the checking of the precondition, stating that “the method *m* may not give a name conflict in the inheritance hierarchy of *C*”.

```
model uml14
package Foundation::Core
context Method
def:
```

...

```
let pre_extract_method(name : String) : Boolean =
  not self.owner.inheritanceTree->exists(class |
    class.getSignatures->exists(method |
      method.name = name
    )
  )
)
```

...

```
let superClass : Set(Classifier) =
  self.generalization.parent.oclasType(Classifier)->asSet

let subClass: Set(Classifier) =
  self.specialization.child.oclasType(Classifier)->asSet

let getSignatures : Set(Operation) =
  self.feature->select(oclasTypeOf(Operation)).oclasType(Operation)->asSet

let allParentClasses : Set(Classifier) =
  self.superClass->union(self.superClass.allParentClasses)->asSet

  let allChildren : Set(Classifier) =
    self.subClass->union(self.subClass.allChildren)->asSet

let inheritanceTree : Set(Classifier) =
  self.allParentClasses->union(self.allChildren)->union(Set{self})->asSet
```

The *Extract Method* refactoring has a rather complex postcondition. Therefore, we decompose it in three constraints:

1. m exists in C ,
2. each local variable, attribute or formal parameter used by the statements is passed to m , and
3. each occurrence of the statements is replaced by a method call of m .

The OCL for (1) is similar to the precondition specification presented above while (2) and (3) require more complex model traversals.

We present the second part of the postcondition in OCL below as it nicely illustrates how *access-*, *update-* and *call-behavior* information can be collected from method bodies. Note that reasoning about such information requires syntactical constructs that are not provided by the standard UML metamodel [268]. Such extensions can be realized by relying on MOF, UML profiles, or a combination of the two techniques.

```
-- oldM: the method from which we have extracted the statements
-- targetM: the method to which we have extracted the statements
let passvariablesasparameters(extrStatements: ActionSequence,
                             oldM: Method,
                             m : Method) : Boolean =
extrStatements.action->select (
  oclIsTypeOf(AccessAction) or oclIsTypeOf(UpdateAction)
).oclAsType(SingleTargetAction).targetRefinement->select(tr |
  ( oclIsTypeOf(LocalVariable) and not (
    ( tr.oclAsType(LocalVariable).isDeclaredIn(
      oldM.bodyRefinement.without(extrStatements)
    ) and not tr.oclAsType(LocalVariable).isUsedIn(
      oldM.bodyRefinement.without(extrStatements)
    )
    ) or tr.oclAsType(LocalVariable).isDeclaredInChildOf(oldM.bodyRefinement)
  )
) or oclIsTypeOf(Parameter)
)->forAll(tr |
  m.parameter->exists(parameter |
    parameter.name = tr.name and (
      parameter.type = tr.oclAsType(LocalVariable).type or
      parameter.type = tr.oclAsType(Parameter).type
    )
  )
) and extrStatements.action->select(ActionSequence)->forAll(childSeq |
  passvariablesasparameters(childSeq, oldM, m)
)

context LocalVariable def:
let isDeclaredIn(as: ActionSequence) : Boolean =
  self.body = as

let isDeclaredInChildOf(as: ActionSequence) : Boolean =
  as.action->select(oclIsTypeOf(ActionSequence))->exists(
    childSeq | self.isDeclaredIn(childSeq
      .oclAsType(ActionSequence)) or
    self.isDeclaredInChildOf(childSeq
      .oclAsType(ActionSequence)))

let isUsedIn(as: ActionSequence) : Boolean =
  as.action->select(oclIsTypeOf(AccessAction) or
    oclIsTypeOf(UpdateAction))
    .oclAsType(SingleTargetAction).targetRefinement
    ->select(oclIsTypeOf(LocalVariable))
    ->includes(self)
```

```

let isUsedInChildOf(as: ActionSequence) : Boolean =
  as.action->select(oclIsTypeOf(ActionSequence))->exists(
    childSeq | self.isUsedIn(childSeq
      .oclAsType(ActionSequence)) or
    self.isUsedInChildOf(childSeq
      .oclAsType(ActionSequence)))

```

Instead of focusing on the UML metamodel extension mechanisms (introduced in [268]), this section aims to illustrate the programming style that is needed to reason about lists of code statements. This programming style resembles pure functional programming in the sense that only side-effect free function calls are involved. On the other hand, the functions traverse an object-graph that represents the model of the source code. Such explicit object graphs are not supported by purely functional programming languages such as Haskell.

Another important observation is that the OCL operation shown for this postcondition may not capture all responsibilities of the method that realizes the *Extract Method* refactoring for a concrete programming language such as Java, or C#. Instead, it models those responsibilities that a transformation designer classifies as essential and subject to potential implementation errors.

3.2.2.2 OCL Code Smell: *Duplicate Code*

One of the goals of the *Extract Method* refactoring is to handle the code smell related to *duplicated code*, as described in [91].

To detect fragments of duplicate code, one needs to reason about the equality of two sequences of statements. This section discusses an OCL expression that supports that task. This expression is also used in the third part of the post-condition discussed in Section 3.2.2.1. Note that more general notions of code duplication [269] may require a more elaborate OCL query.

In the example two sequences are given together with two integers, denoting the entries in each sequence that are compared with each other. The OCL code relies on a helper operation called *equalsByValue* that has not been defined in the standard but can either be generated for any UML metaclass or can be realized by using OCL introspection.

```

let check_statement_match(src : Sequence(Action),
  i : Integer,
  extract : Sequence(Action),
  j : Integer) : Boolean =
  if j >= extract->size then
    true
  else
    if src->at(i).equalsByValue(extract->at(j)) then
      check_statement_match(src, i+1, extract, j+1)
    else
      false
    endif
  endif

-- shorthand for client call
let check_stmt_match(src: Sequence(Action), extract: Sequence(Action)): Boolean =
  check_statement_match(src, 0, extract, 0)

```

This fragment provides another example of the “functional programming style” (or “the use of recursion”) for list processing that is imposed by the side-effect free nature of the OCL. It should be noted that OCL also supports another programming style that is still side-effect free, yet resembles imperative loop programming:


```

let check_statement_match_LOOP (
  src : Sequence(Action),
  extract : Sequence(Action)
) : Boolean =
src->size = extract->size and
Sequence(1..src->size)->forall(i|
  src->at(i).equalsByValue(extract->at(j))
)

```

Clearly, the choice for a particular specification style impacts the readability of the OCL expressions. Section 3.3.1 illustrates another driver for a particular specification style: some OCL expressions can be automatically analyzed and transformed while others cannot.

3.2.3 Pull Up Method

The “Pull Up Method” refactoring moves a method m with implementation MD of a class C into its superclass SC . After this first move, all methods carrying the same signature in C ’s sibling classes can be removed. The aim of this refactoring is to centralize functionality in a common super class.

The precondition of the *Pull Up Method* refactoring can be decomposed in four constraints:

1. SC must exist, i.e., C may not be a root class,
2. the signature m may not already exist in superclass SC ,
3. the implementation of m shouldn’t refer to any variables defined in C , and
4. the implementation of m shouldn’t refer to a method not accessible in SC .

The following fragment illustrates how this decomposition can be expressed naturally in OCL, by relying on helper operations to hide the complexity of the individual constraints:

```

let subClass: Set(Classifier) =
  self.specialization.child.oclAsType(Classifier)->asSet

let getSignatures : Set(Operation) =
  self.feature->select(oclIsTypeOf(Operation)).oclAsType(Operation)->asSet

let allParentClasses : Set(Classifier) =
  self.superClass->union(self.superClass.allParentClasses)->asSet

```

The postcondition and code smells related to this refactoring can be found in [268].

3.2.4 Discussion

We have shown that it is feasible to use OCL to describe refactoring contracts on a meta-model that hides some language-specific language constructs into *access*-, *update*-, and *call*-constructs. Still, the OCL expressions are highly technical and require an in-depth knowledge of refactoring. It should be stressed that the use of OCL does not automatically make the contract implementations more readable than an implementation in a general purpose programming language.

From a pragmatic programming perspective, the ability to define helper operations is essential to manage the size of, and the reuse across, refactoring contracts. Obviously, such

helper operations can easily be realized in general purpose programming languages too. Coming to the more subtle consideration of the proposed programming paradigm, one can remark that the side-effect free nature of the OCL forces developers (or maintainers) of imperative refactoring implementations into another mindset. This may lead to the discovery of hidden errors.

More specifically, the loose coupling between a transformation contract and the corresponding constructive model of the behavior enables one to consider the same problem from different perspectives. This may increase the potential of catching exceptional cases in the testing process. We are unaware of empirical studies that investigate the probability that developers make the same mistakes twice when specifying the same problem in two different paradigms.

However, if this probability is low enough to compensate the cost of the redundant specification, one should discover a significant amount of errors by specifying a refactoring's post-condition in a side-effect free language such as the OCL and by realizing the actual behavior in an imperative programming language such as Java or Story Diagrams. Controlled experiments should be conducted to measure the specification cost and the amount of errors caught.

The usefulness of specifying refactorings is not limited to finding errors during transformation development. More specifically, Section 3.3.1 illustrates how the use of a side-effect free (or at least "restricted", as opposed to "general purpose") language such as the OCL can support the development of composite refactorings.

3.3 Applications of Automated UML Refactoring

This section illustrates how OCL refactoring enable the automatic application of refactorings to specific code fragments.

Subsection 3.3.1 describes what kind of automatic compositions one may expect from a refactoring tool. Although the OCL was not designed with such compositions in mind, the case study is an interesting validation for ongoing work in the area of OCL based model checking. Subsection 3.3.2 illustrates how refactoring operations can be selected automatically based on OCL helpers describing bad code smells.

3.3.1 Compose primitive refactorings

Don Roberts has made an initial analysis of *composite refactorings* (i.e., sequences of more primitive refactorings [217]). More specifically, he provides an algorithm to compute the pre-conditions for a composite refactoring from its primitive constituents. The chain of primitive constituents needs to be provided explicitly by the transformation writer. In this section however, we investigate whether OCL refactoring contracts can be used to derive chains of primitive refactorings automatically.

To motivate the need for automatically computing such chains, we present a concrete example that involves the refactorings presented in the previous section.

Consider the class hierarchy displayed on Figure 3.1 (a): two classes, *SubclassB* and *SubclassC*, with a common superclass, *SuperclassA*, both have a method *foo* with almost the same implementation. The method *foo* should be pulled up into the superclass. However the pre-condition of the *Pull Up Method* refactoring as described in Section 3.2.3 is violated. The two

Figure 3.1: Application of *Extract Method* to problematic class hierarchy.

conditionals (“`a==b`” and “`a==c`”) do not match, so the implementation of the methods is not identical.

However if we consider other primitive refactorings, such as *Add Method*, as described by Opdyke and Roberts it is possible to find a sequence of refactorings that solve this mismatch and whose pre- and postconditions guarantee behavior preservation [202, 217].

When we extract the conditional in both classes to a method with a name *bar* using *Extract Method*, the postcondition of this refactoring ensures that each occurrence of the statements is replaced by a method call to *bar*. This results in two identical implementations of *foo* in *SubclassB* and *SubclassC*. This situation is visualized by the class diagram shown on Figure 3.1 (b).

Still, the *Pull Up Method* cannot be performed yet. The two implementations of *foo* access a method (*bar*) not defined in the superclass *SuperclassA*, which violates the precondition of *Pull Up Method*. Combining the pre- and postconditions of the *Extract Method* refactoring, we know that *bar* does exist in *SubclassB* and *SubclassC*, but nowhere else in their inheritance tree. This satisfies the precondition of the *Add Method* refactoring applied on *SC* that requires that there is no method with the same signature as *testCondition* present in superclass *SuperclassA*. After applying this *Add Method* refactoring, its postcondition guarantees that *bar* exists in *SuperclassA*. This situation is visualized by the class diagram shown on Figure 3.2 (a).

The postconditions of the *Extract Method* and *Add Method* refactorings satisfy the precondition of *Pull Up Method*, hence this refactoring can be applied. The resulting class hierarchy is displayed by Figure 3.2 (b).

Coming to the OCL refactoring contracts proposed in this chapter, reengineering tools can automatically verify whether all refactorings in one particular chain have satisfied preconditions. What hinders complete automation of high level refactoring compositions however, is the lack of an inference engine that can efficiently select and parametrize the refactorings being composed.

In terms of the presented example, an engine should automatically relate the *Extract Method* and *Add Method* refactorings to the *Pull Up Method* refactoring. Then, it should efficiently select the statements at the right of the two equality operators (within the conditional) as can-

Figure 3.2: Application of *Add Method* and finally the intended *Pull Up Method*.

didates for being extracted. This presents several conceptual and computational challenges:

1. The refactoring engine should be able to relate the postconditions of the *Extract Method* and *Add Method* refactorings to the precondition of the *Pull Up Method* refactoring automatically. This requires an OCL subsumption analysis, while the semantical basis supporting such an analysis is not provided by the OCL standard.
2. An exhaustive search for possible refactoring chains is computationally not feasible. It should be clear that a refactoring engine should not explore just any application of the *Extract Method* refactoring in trying to create a match between the two implementations of the method that needs to be pulled up. Instead, the engine should detect the differences between the method bodies and focus on searching a way to remove those differences.

At the time of first publishing this work [268], the OCL did not provide any support for tackling these challenges. However, recent research focuses on translating OCL into the languages of model checkers. This provides a promising basis for automating the composition of refactorings since such languages do have a well-defined notion of constraint subsumption. However, the current state-of-the-art at least imposes some syntactical restrictions. For example, Cabot's approach to translate OCL to Constraint Satisfaction Problems [38] would forbid the use of recursion in the helper operations that are called from refactoring contracts.

3.3.2 Integrate with code smell detectors

In this section, we demonstrate how OCL refactoring contracts can be applied for the semi-automatic application of refactorings. It has already been demonstrated that OCL is adequate to specify design metrics on the UML metamodel [14].

In Section 3.2 we showed how our extended metamodel enables us to collect program information from inside the method body and describe the code smells from Fowler in our refactoring contracts [91]. The next step is to agree on a threshold value that says when the code smells so bad that it needs to be refactored. The risk thresholds published by the

Software Engineering Institute [121] can be used as initial values but they are likely to be aligned with each project's priorities. These thresholds can be stored in the context of the refactoring contracts for a specific language or application.

Applying the logical "AND" from the OCL code smell with the OCL precondition of the same contract, to a model of the source code, results in a list of methods that

1. are *desired* to be transformed – as they exceed the threshold,
2. are *allowed* to be transformed while preserving their behavior – as the precondition of a refactoring that solves the problem is satisfied.

After applying the refactoring the model can be validated using the OCL postcondition from its contract.

3.4 Related and Ongoing Work

Fowler's refactorings are specified as source-to-source transformations [91]. The concept of model-driven refactoring has historically been investigated in the context of *design model* restructuring. Sunyé for instance explored how the integrity of class diagrams and statecharts can be maintained over refactoring [244]. The authors use the plain UML metamodel to describe refactoring pre- and postconditions in OCL. Due to the limitations of the UML 1.4 metamodel, however, they are unable to formalize the relation between the class (i.e., structural) and statechart (i.e., behavioral) diagrams. An extended metamodel such as the one used in this chapter can solve this problem.

One of the contributions of this chapter is the notion of a *refactoring contract*. Opdyke was one of the first to describe primitive refactorings together with the *preconditions* that must be met to ensure that the transformation preserves program behavior [202]. Roberts extended this work by augmenting the refactoring definitions with *postconditions* and showing how refactorings can be composed [217]. Tourwé and Mens use logic metaprogramming to integrate *code smells* in an automatic refactoring tool [251]. Prolog-like logic rules are used to detect bad smells in the program code and to suggest refactorings that can be used to remove these smells. We have raised the abstraction level by using a metamodel that represents a wide range of object-oriented languages instead of using a Smalltalk specific abstract syntax tree directly. Moreover, we have investigated to what extent the OCL is suited to formalize the contract definitions.

To reason about a chain of refactorings, the metamodel used to represent source code needs to be extended with versioning information or evolution information. Mens et al. have proposed preliminary UML extensions for this purpose [161, 159] but more work is needed to determine how snapshots from a sequence of refactorings can be represented formally. Markovic proposed to represent all variables representing such a snapshot explicitly with an index representing the order in the chain but can only handle primitive variable types [153]. Unfortunately, a more powerful, metamodel based, approach has not been developed yet.

3.5 Summary and Outlook

In this chapter we have evaluated to what extent the OCL is useful for formalizing refactoring contracts. Such a refactoring contract consists of (i) a precondition that states when the refactoring can be triggered without introducing undesirable side-effects, (ii) a postcondition that describes the effect of the refactoring and (iii) a code smell that describes when the refactoring can improve a particular (set of) quality attribute(s).

Several OCL fragments related to two representative refactorings illustrate that the side-effect free nature of the OCL involves a programming style that differs significantly from the imperative style needed to implement a refactoring. Empirical experiments are missing to assess whether thus approaching the same problem from different perspectives efficiently leads to less errors in the resulting refactoring tool.

However, specifying refactoring contracts in OCL may be a promising basis for realizing the automatic composition of refactorings in the long term. In the short term, OCL refactoring contracts can already be used to integrate refactoring tools with OCL based metric suites such that refactorings can be triggered automatically when a particular quality threshold has been passed.

Obviously, an OCL refactoring contract does not model how actual model elements need to be transformed. More specifically, a refactoring contract is a *restrictive* transformation specification: the contract only states what needs to be changed (cfr., the postcondition) in which case (cfr., the precondition) and to which aim (cfr., the code smell). The next chapter illustrates how other UML language elements can be used to model the transformation behavior *constructively*, in an object-oriented manner.

Chapter 4

Expressing Refactorings as Story Diagrams

This chapter is based on a paper that has been published in the proceedings of the Fujaba Days in 2003 [258]. The chapter illustrates how the actual behavior of a refactoring can be modeled in a particular UML dialect, called *Story Diagrams*. More specifically, the *Pull Up Method* refactoring is implemented using – *and deployed to* – the primary tool for Story Driven Modeling, called *Fujaba* [100, 200]. The chapter applies the framework of “cognitive dimensions” to illustrate how Story Diagrams fit several requirements for modeling model transformations. Subsequent chapters will revisit this analysis for aligning the design of the language even more with model transformation requirements.

This chapter is structured as follows: Section 4.1 comes back to the purpose and nature of the “Pull Up Method” refactoring, which has already been discussed in Chapter 3. Section 4.2 presents how the behavior of that refactoring can be modeled with Story Diagrams. Section 4.3 contains the cognitive evaluation of the transformation model while Section 4.4 shows how several implementation specific artifacts can be generated automatically from the transformation model. Finally, Section 4.5 concludes the chapter.

4.1 Case Study: Pull Up Method

The main purpose of refactoring is to prepare an existing object-oriented system for the addition or correction of some behavior. The “Pull Up Method” (PUM) refactoring operation that is discussed in this section supports this purpose by moving some functionality to a more reusable location. More specifically, the PUM refactoring moves a method from a subclass to a (presumably more generic) superclass. From that location, this method can be used directly from new sibling classes while it is overridden by methods with the same signature in existing sibling classes.

Figure 4.1 shows a Fujaba screenshot that illustrates how the PUM refactoring can be used to pull up the *handle* method from *Workstation* class to the *Node* class. Note that the *Printserver* class would then specialize this method while new subclasses of *Node* would share the *handle* functionality with the *Workstation* class. Another well-known variant of the PUM refactoring not only moves a method to the superclass but also removes methods with

Figure 4.1: Application of the “Pull Up Method” refactoring on the method *handle* of the class “Workstation”.

the same signature from sibling classes [92].

Remark that it does not make sense to move the *printASCII* method from the class *Printserver* to its more generic superclass *Node*. In contrast, the refactoring does make sense for the *handle* method of class *Workstation*. Thus, the application of the *Pull Up Method* refactoring needs to be controlled explicitly by a modeler. As shown by the screenshot from Figure 4.1, the refactoring may be triggered by right-clicking on a particular method. As discussed in the previous chapter, the tool should only execute the refactoring when its precondition is satisfied.

The subsequent sections will illustrate how the Fujaba plugin related to this refactoring functionality has been developed in a model-driven manner.

4.2 Transformation Modeling using Story Diagrams

For the sake of conciseness and focus, this thesis will not present any Story Boards that could have been used to derive Story Diagrams. In fact, the systematic use of Story Boards in a model transformation context was hindered by missing fragments in the tool chain used. More specifically, no stable JMI plugin for Fujaba’s *Dynamic Object Browsing System* was available [90]. Since we do not want to draw wrong conclusions due to some bugs, the remainder

Figure 4.2: Model transformation for the PUM refactoring.

of this thesis will focus on Story Diagrams.

Subsection 4.2.1 presents how the core behavior of the Pull Up Method refactoring can be modeled as a very simple Story Diagram. Subsection 4.2.2 presents how the precondition can be modeled with Story Diagrams too.

4.2.1 Model Transformation

The behavior of the actual transformation can be expressed with one simple graph transformation rule. This rule presumes that the precondition of the refactoring has been checked beforehand. Under these circumstances, it is sufficient to remove the method from its current class and associate it with the superclass. Figure 4.2 shows how this can be expressed as a Story Diagram.

In the upper state, the rule checks whether the *method* instance that is passed as an argument to the transformation is in fact a proper *UMLMethod*. The rule removes the link to the current class, as specified by the *«destroy» methods* edge between the method and its containing class. The right part of the state shows a pattern between this class and its superclass. The *«create» methods* edge creates a link between the method node and that superclass.

4.2.2 Precondition

Next to specifying the actual model transformation, it is also possible to model the precondition of the PUM refactoring as a Story Diagram.

Modeling this precondition in an understandable manner required a small extension to the language for creating Story Diagrams [258]. More specifically, it required a construct to navigate across ordered association links. However, to illustrate that Story Diagrams were already capable of expressing a large part of the precondition, Figure 4.3 displays the original version of the precondition model.

The first state of the Story Diagram checks whether the argument instance, named *method*, is contained in a class that has a superclass: obviously, without such a superclass, the method cannot be moved higher in the inheritance hierarchy. The outgoing transition with the *[failure]*

Figure 4.3: Story Diagram that models the precondition of the Pull Up Method refactoring.

label is triggered when this pattern from the method to the superclass cannot be matched. The transition leads to an end state returning *false* to indicate that the precondition is not satisfied.

The transition labeled with *[success]* is triggered when the superclass does exist. In that case, the transformation continues in the state holding the *superclass* node, that was already bound in the previous state, and a yet to be bound *methodFromSC* node of type *UMLMethod*. The latter node is bound by matching across the *methods* association. The double edge around the state indicates that this state realizes an iterative loop. Thus, it is executed as long as new matches can be found. Consequently, the state is used to find all the methods from the superclass.

The outgoing transition labeled with *[each time]* ensures that for each such method, the rule checks whether it has the same signature as the method that should be pulled up. If this two-step check succeeds, the rule has found that the superclass already has a method with the signature of the method that should be pulled up. In that case, the precondition fails, since pulling up the method would result in two methods for the same message, which violates the Java semantical constraints.

Note that the comparison of the signatures of the two methods is implemented by first checking the equality of their names and then checking whether their parameters match. As soon as a comparison fails, the transformation returns to the state that selects methods from the superclass. When all such methods have been matched without finding a signature clash, the transition labeled as *[end]* is triggered. In that case, the Story Diagram indicates that the precondition is satisfied by returning *true*.

Coming to the limitations of the notation presented in Figure 4.3, note that the *checkParametersMatch* method was first implemented in Java due to the lack of a construct for indexing an ordered association link. Figure 4.4 shows that in practice this could easily be

Figure 4.4: Integrating Java Code into the PUM sample Story Diagram.

Figure 4.5: Metamodel structure related to the signature of a method.

solved by leveraging the ability to insert regular Java code into Story Diagrams.

Still, since this Java code exhibits the classical problems associated with textual programming, we aimed for a better representation of this behavior. The solution can be derived from an investigation of the metamodel structure. The relevant fragment is shown on Figure 4.5.

This diagram shows that the ordered association from *UMLMethod* to *UMLParam* can be qualified with the *index* property of the contained elements. Therefore, it makes sense to provide an equivalent visualization to actually index this association at runtime. The bottom of Figure 4.6 leverages a visual notation to index the *param* association between *m2* and *p2* (whose meaning is described below). Another visualization, that lowers the mental gap between the metamodel and the constraint specification even more, would put the *p1.getIndex()* inside a box next to *m2*, thus mimicking the qualified association from the metamodel.

By creating a correspondence between the metamodel syntax (class diagrams) and the model transformation and analysis syntax (Story Diagrams), one benefits more from the visual clues that are made explicit in the metamodel. As a practical example of why this is useful, note that by investigating the metamodel, one can easily consider all associations that attach to *UMLMethod*. Such a metamodel investigation could enable one to discover that the code shown on Figure 4.4 did not take into account that the signature of UML methods is also defined by the result type of the methods.

Checking the equality of the return types of the two methods is part of what is done in the first state of the Story Diagram shown on Figure 4.6. Additionally, the state checks whether *m1* and *m2* have the same name. Checking the number of parameters associated

order to optimize the quality of their models.

When considering the Story Diagram from Figure 4.2, it should be noted how the Story Diagram syntax enables one to specify the complete transformation in “one picture”. More importantly, all relations between these elements are displayed in a compound diagram, that is still manageable in size. The contrast to the corresponding source code is remarkable:

```

13  public void execute(ASGElement target)/* id=id9128    # no Name */
14  {
15      boolean fujaba__Success = false ;
16      Iterator fujaba__IterContainerRevSubclassStub = null ;
17      UMLClass container = null ;
18      UMLClass superclass = null ;
19      UMLGeneralization stub = null ;
20      UMLMethod method = null ;

    ...

32  method = (UMLMethod) target ;

    ...

35  container = method.getParent () ;

    ...

39  fujaba__IterContainerRevSubclassStub = container.iteratorOfRevSubclass () ;
40  while ( !(fujaba__Success) && fujaba__IterContainerRevSubclassStub.hasNext () )
41  {

    ...

44  stub = (UMLGeneralization) fujaba__IterContainerRevSubclassStub.next () ;

    ...

48  superclass = stub.getSuperclass () ;
49  JavaSDM.ensure ( superclass != null ) ;
50
51  // check isomorphic binding
52  JavaSDM.ensure ( !(container.equals (superclass)) ) ;
53
54  // delete link
55  container.removeFromMethods (method);
56
57  // create link
58  superclass.addToMethods (method);
59
60  fujaba__Success = true ;

    ...

65  }

    ...

79  }

```

Although this source code *fragment* can also be shown on a limited amount of space, the relations between the transformed model elements are undesirably implicit. The lookup of the container class and its superclass is separated from the relocation of the method element. This is done for operational purposes: the superclass cannot be found before the container class has been matched, etc. The implicitness of the relationships is undesirable since one needs to build a mental model of the expected input model based on statements that have been ordered in a machine-oriented order.

For example, in order to understand the meaning of “superclass” on *line 58* one needs to scan through lines 52 and 48 to learn about the relation between *superclass* and *stub*, and scan to *line 44, 40 and 39* to learn about the relation between *stub* and *container*. Finally, one would need to scan to *line 35* to learn about the relation between *container* and *method*. Keeping all this information in mind, one needs to create a mental picture of the path from *superclass* to *method* in order to understand to what element the *method* element will be moved to. Thus, Green’s “role-expressiveness” dimension has not been taken into account at the source code level, forcing “hard mental operations” (the scanning of source lines).

The Story Diagram from Figure 4.2 on the other hand does not force the problematically operational mindset. Instead, one may focus directly on the relocation of the method from one class to another one. From there, one notices that these two classes should inherit from one another, but the specification does not state which element is looked up first, as it is irrelevant for understanding the transformation.

Thus, the Story Diagram eliminates some of Green’s “hard mental operations” by making “hidden dependencies” explicit. Moreover, when writing the Story Diagram from Figure 4.2, one does not have to think about the order between particular statements, thus taking Green’s “premature commitment” factor into account.

Other cognitive observations can be made about the Story Diagrams of the Precondition, as shown on Figure 4.3 and 4.6. For example, in the latter Story Diagram, the *returnType* variable is shown only once, and is explicitly linked to both method variables. This explicitness is absent in the Java code derived from the Story Diagram. A major advantage of visual modeling is its support for a “secondary notation”. Green summarizes this dimension by means of the following question [106]: “Can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the official semantics of the language?”

In the case of Story Diagrams, a transformation modeler has the freedom to exploit 2D layout patterns for conveying conceptual relations between model elements. For example, across the entire Story Diagram shown on Figure 4.6, method *m1* is handled on the left side of the diagram while *m2* is handled on the right side. This supports the semantical clue that these methods are being compared side by side. Such clues cannot be given when using a textual programming language.

Coming back to the elimination of “hard mental operations” (step-wise scanning of source lines), note that same Story Diagram has only two final states, corresponding exactly to the only two possible return values of the transformation: *true* and *false*. These exit points are shown on opposite sides of the diagram: *true* is shown on the left, while *false* is shown on the right. To investigate in what cases the method returns that the parameters of the two argument methods do not match, one can easily follow all transitions leading to the final state that is labeled *false*:

- when *m1* and *m2* have a different return type (transition 1), or
- when *m1* and *m2* have a different amount of parameters (transition 2), or
- when a parameter *p1* from *m1* has a different type than the corresponding parameter of *m2*.

Deriving this information from the corresponding source code would force one to look for *all* statements returning *false*. Without an equivalent of the explicit transition links, this is an error-prone task.

As a limitation of the techniques presented in this section, consider the specification of the types of parameter $p1$ and $p2$ within the Story Diagram shown on Figure 4.6. Ideally, one would model that $p1$ and $p2$ are related to the same node, representing the shared parameter type. However, as links cannot cross the boundaries of one Story Pattern, $p2$ from the Story Pattern on the bottom right of the Story Diagram from Figure 4.6 cannot point to the $t1$ that is matched in the Story Pattern on the bottom left of that Story Diagram. Therefore, $t1$ is replicated as a bound node within the Story Pattern on the bottom right.

In principle, one could create an additional view upon that former Story Pattern, where the path from the already bound $m1$ and $p1$ nodes to the $t1$ node would be shown redundantly next to the path coming from the already bound $m2$ node and the $p2$ node being matched. This view would visualize all elements related to the definition of a parameter type match between the “left” ($m1$) and “right” ($m2$) branches in one diagram, thus supporting Green’s “role expressiveness” dimension again. Such views on Story Patterns however are not supported by the main Fujaba release and will be introduced in subsequent chapters, where a minimalistic Fujaba extension will be discussed.

4.4 Deploying and Using the Transformation

The previous section motivated that the use of Story Diagrams is preferable over conventional source code from a usability perspective. The use of an explicit transformation model also enables one to derive other essential code artifacts that are undesirably platform specific. More specifically, a refactoring implementation needs to be complemented by an XML plugin descriptor before it can be deployed to the Fujaba tool. In the context of the Fujaba refactoring prototype, this plugin descriptor has been written by hand. In the following chapter, a tool will be presented that generates such descriptors from the transformation model too.

The resulting refactoring can be invoked from UML diagrams: in fact, Figure 4.1 from Section 4.1 shows the application of the plugin on a UML class diagram that models an application for LAN simulations. The screenshot shown on Figure 4.1 is taken from the following round-trip engineering cycle: first of all, existing Java source code files are imported into Fujaba. Within this UML tool, the re-engineer investigates the class hierarchy in order to detect some design problems.

As soon as such problems have been detected, (s)he applies refactorings such as the “Pull Up Method” to change the structure of the system under study. Finally, the Java code is regenerated from the UML models. Figure 4.7 summarizes this development cycle by presenting screenshots from the Fujaba tool. The screenshot from the third step corresponds to the one shown on Figure 4.1.

4.5 Summary and Outlook

This chapter presents a transformation model of the Pull Up Method refactoring as a set of Story Diagrams. An analysis along the “cognitive dimensions” framework of Green confirms that Story Diagrams are a promising formalism for modeling model transformations.

The most obvious advantage of visual modeling over textual programming is the elimination of the “hard mental operations” that are related to manually scanning source code lines. In textual source code, such scanning is often needed to check what other elements are related

Figure 4.7: Round-tip engineering cycle in which the “Pull Up Method” refactoring is applied.

to a particular model element. The need to choose exactly one sequential decomposition of all statements related to matching a compound pattern puts an undesirable emphasis on the elements that happen to be matched for technical reasons.

Another scenario that requires one to scan source lines manually is the lack of an explicit link between all statements that return a transformation’s result. Visual languages such as Story Diagrams solve such problems by means of explicit edges that are unrestricted by the formatting constraints of text-based editing. Another major advantage of Story Diagrams is the ability to exploit layout patterns as a “secondary notation”.

Apart from these confirmations that the use of Story Diagrams makes sense, the analysis proves its value by indicating a potential improvement of the formalism: the “role expressiveness” of Story Patterns can be improved by supporting redundant views upon such diagrams. This concern will be addressed in the subsequent chapter and subsequent chapters will illustrate the value of transformation views on several examples.

Finally, the chapter indicated that transformation models can also be used to hide platform-specific details by means of code generation. This topic will be investigated in detail in the next chapter.

Chapter 5

Standardizing Story Diagrams using UML and MOF

The contributions from this chapter have been published in the proceedings of the *Fujaba Days* and *SETra* workshops in 2004 [225, 226]. I have identified the underlying problems in the context of version 3 of the Fujaba tool. However, these problems are still relevant today and the proposed solutions are independent of the Fujaba tool.

The previous chapter illustrated how *Story Diagrams* [86, 282], Fujaba's UML based graph transformation language, can be used to develop refactoring plugins for the Fujaba tool in a model-driven manner. However, there are two major problems that may hold the MDA community from adopting that approach in a more general transformation development context.

First, Fujaba's Story Diagrams are based on a proprietary metamodel that is in turn based on a proprietary meta-metamodel. In practice, this means that transformations that are modeled as Story Diagrams cannot be edited with other tools. Moreover, this means that this exchangeability problem is hard to overcome since the proprietary metamodel is only implicitly present in the Fujaba source code.

Secondly, due to assumptions of the code generator, the generated transformation code can only be deployed to the Fujaba tool itself. The cause of this problem is that the Fujaba code generator only integrates with code complying to a Fujaba specific API. More specifically, it generates code that is based on a Fujaba-specific association framework [94].

In this chapter, we describe how such issues can be overcome by using a flexible code generator for translating transformation models that conform to a standard such as UML to complete model transformation code that complies to a model management standard such as the Java Metadata Interface (JMI [168]). This proposed architectural solution is supported by a tool prototype called MoTMoT [225, 226]. This chapter illustrates how this approach solves the problems outlined above.

More specifically, the refactoring from the previous chapter is modeled in one UML tool and the generated plugin is deployed to another UML tool. Subsequent chapters will rely on the same infrastructure for realizing other types of transformations. These chapters illustrate the proposed approach as well as the supportive prototype are independent of specific metamodels or transformation types.

Fujaba’s repository sources. The class diagram highlights the first problem: although Fujaba’s metamodel does *resemble* a standard metamodel, that is: the UML 1.3 metamodel, it does not actually share many similarities at the detailed level. Moreover, it turns out that Story Diagrams are realized by means of dedicated metaclasses. These metaclasses are completely Fujaba-specific which means that no other tool is able to read or write models of that form.

To overcome this problem, the following subsection presents an OMG UML 1.4 compliant profile for Story Diagrams.

5.1.2 Solution: UML Profile for Story Diagrams

In summary, designing a UML profile comes down to *mapping* each new language construct to an existing UML counterpart. Additionally, stereotypes are used to differentiate between several variants of a UML construct. For example, an *iterative loop* state should be represented by a different kind of UML activity as a normal state in a Story Diagram.

Mapping the the control flow part of Story Diagrams is rather straightforward. Essentially, the *ActionState* metaclass is used as the basis for all Story Diagram activities. An iterative loop is realized by means of the «loop» stereotype, arbitrary Java code can be triggered from states carrying a «code» stereotype, while direct calls to other Story Diagrams are supported by means of the «link» stereotype. The «success» and «failure» stereotypes can be used on transitions that leave an activity that is modeled by a Story Pattern, while the «each time» stereotype can be used to define the body of a loop.

Note that within a transformation model, not every aspect needs to be modeled using Story Diagrams. Instead, only methods that are decorated using the «ModelTransformation» stereotype are modeled using Story Diagrams. The relation between such a method and the corresponding activity diagram is realized by means of the “motmot.transformation” tagged value. This tag should point to the package that contains the activity diagram modeling the transformation flow.

Story Patterns, or so-called “*transformation primitives*”, are represented by decorated *class diagram* elements, although the implicit Fujaba metamodel relies on *collaboration diagram* elements. This design choice is motivated by several observations. First of all, standard class diagrams offer more explicit visual features than standard collaboration diagrams. For example, attribute assignments can be represented intuitively using the initial value property of UML attributes.

Similarly, class diagrams show cardinalities and role names on association ends, while this information cannot be shown in a collaboration diagram. Since this information is quite useful for understanding the relation between a particular rewrite rule and the metamodel of the transformed graphs, it makes sense to show it explicitly.

Finally, a pragmatic argument for relying on class diagram elements instead of collaboration diagram elements is that the former diagram type is supported by more UML tools than the latter diagram type. Therefore, relying on class diagrams increases the exchangeability of Story Diagrams.

Thus, the link from the activity diagram representing a transformation’s control flow and the corresponding Story Patterns is realized by associating an *ActionState* with a class diagram. More specifically, the UML profile for Story Diagrams provides a tagged value “motmot.transprimitive” on *ActionState* that has type *Package*. This package should contain the class diagram that defines a particular Story Pattern. Upcoming chapters will illustrate that such a package can actually contain several class diagrams, that define overlapping views upon

a complex Story Pattern. Additional application conditions of a rewrite rule can be encoded by instantiating the “*motmot.constraint*” tag on an *ActionState*.

Similar to the original Story Diagram syntax, the UML profile enables one to show the name of the nodes being matched on the left-hand side of a colon while the node type is shown on the right-hand side. The limitation of this approach is that both the node name and node type information is entered in a field of type String. When writing transformations of models conforming to large metamodels (such as OMG UML), some metaclass names occur in several packages. In these cases, the node type at the right of the colon may need to be a lengthy fully qualified name to avoid ambiguities. Moreover, inconsistencies arise when changing the names of metaclasses (due to metamodel evolution) since there is no formal coupling between the strings within the Story Pattern nodes and the names of the corresponding metaclasses. To overcome these problems, the UML profile for Story Diagrams also enables one to link a rewrite node explicitly to a metaclass by means of the tagged value “*motmot.metatype*”. When updating the name of a metaclass, references to that metaclass will not become invalid, since the reference is realized by means of a pointer (e.g. an XMI *idref*) instead of by a hard reference to the old name value.

Nodes that are already bound are decorated with the «bound» stereotype. Finally, transitive link patterns can be specified by means of the «closure» stereotype. The core constructs of the UML profile for Story Diagrams are summarized in Table 5.1.

SDM Construct	UML Construct	Related Profile Elements
Method	Method	«ModelTransformation», <i>motmot.transformation</i> tag
Story Diagram	Package, Activity Diagram	<i>motmot.transformation</i> tag
Iterative Loop	State Transition	«loop», «each time»
Conditional	Transition	«success», «failure»
Textual State Constraint	State, String	<i>motmot.constraint</i> tag
Call	State, String	«link», «code»
Story Pattern	State Package, Class Diagram(s)	<i>motmot.transprimitive</i> tag <i>motmot.transprimitive</i> tag
Node / Object	Class	<i>motmot.metatype</i> tag, «bound», «create», «destroy»
Textual Node Constraint	Class, String	<i>motmot.constraint</i> tag
Attribute Assignment	Attribute (initial value)	
Edge / Link	Association	«closure», «create», «destroy»

Table 5.1: A UML Profile for Story Diagrams

5.1.3 Discussion

In order to represent Story Diagrams in terms of standard UML language constructs, some subtle syntactical differences need to be introduced. As an initial and brief discussion of the proposed profile, this section assesses the impact of such differences on the understandability

of the modeling language. Subsequent chapters revisit this analysis in terms of their concrete case studies. Concrete differences are already visible between the Story Diagram example in this chapter (cfr. Figure 5.5 on page 99) and that from the previous chapter (cfr. Figure 4.2 on page 81).

First of all, remark that there is one remarkable difference between the representation of Story Diagrams within the “native” Fujaba editor and that within a general purpose UML editor: while the Fujaba editor supports the embedded representation of Story Patterns within Story Diagram states, a general purpose UML editor represents such elements in two strictly separate layers. The impact of this difference can be assessed by means of Green’s framework, which has been discussed in taxonomy section 2.4.2.

More specifically, the *visibility* dimension is primarily affected by this change in representation: the native Fujaba representation gives “complete pictures” of all aspects of a Story Diagram. Both the control flow and rewriting aspects of transformations are shown on compound diagrams. On the other hand, the profile representation positively impacts the *secondary notation* dimension: by not displaying the Story Pattern elements within the state elements, additional space becomes available to provide a description of the state’s intent in natural language.

From a more tool-oriented perspective, the *error-proneness* dimension of the modeling language is also affected: when using a general purpose UML editor, there is less “correctness by construction”. More specifically, as elements only get meaning after attaching specific stereotypes and tags in the profile approach, one can unfortunately forget such decorations. Similarly, modelers are able to combine the stereotypes and tags in ways that do not make sense at all. In contrast, when using the native Fujaba editor, a modeler would for example never be enabled to incorrectly decorate a «code» state as a «loop» too.

Fortunately, the UML profile mechanism provides measures to counter the problem described above. First of all, stereotypes are typed with a particular base class (as discussed in Section 1.2.1.3 from the introductory chapter). This mechanism is used to ensure that, for example, the «loop» stereotype is not applied on transition, node, or edge elements. Secondly, more semantical checks are expressed by OCL well-formedness rules. Thus, generic UML tools can be extended with a Story Diagram specific semantical analyzer only by loading profile constraints that have been defined in a tool-independent manner [224].

Subsequent chapters will illustrate several advantages of the profile approach. In a nutshell, Chapter 6 will illustrate how the profile supports the definition of views on Story Patterns. Moreover, that chapter highlights how the profile enables one to reuse Story Pattern definitions across states from potentially different Story Diagrams.

5.2 Need for executability on Standard Repositories

In spite of Fujaba’s value in validating numerous model management techniques [184, 183, 275], the tool lacks standardization at the execution level of the graph transformations. As stated above, its code generator not only reads its input from a non-standard repository, it also generates output code for the same kind of repositories: again, it relies on its proprietary API design. In what follows, Fujaba’s API design is compared with the Java Metadata Interface standard that was discussed in Section 1.2.3.2 from the introductory chapter.

5.2.1 Problem/Limitation: Fujaba Association Framework

The Fujaba Association Framework [94] is one of the first bidirectional, auto-consistent, collection implementations. Although alternatives have been proposed over time, its core features have remained stable:

- bidirectional reference navigability for associations of any type (*M-to-N* with *M* and *N* being either *1* or ***), and
- integrity preservation when adding or deleting elements on one reference side only.

Remarkably, even in 2004 [150], these features were not yet provided by all major CASE tools and active research was conducted to:

- match the mental model of application developers,
- support maintenance tasks for tool developers.

The work of this thesis is driven by other design goals:

- transform models conforming to any metamodel,
- load/store models from/to XMI,
- manipulate models in-memory, using a standard (i.e., JMI based) CASE tool API.

Although the Fujaba code generator could be used to generate repositories for any metamodel, these repositories would not be able to load and store XMI. Additionally, the APIs of these repositories would comply to the Fujaba Association Framework instead of to the Java Metadata Interface, thus complicating the in-memory integration with other repositories that do comply to this MDA standard.

Fortunately, the NetBeans MetaData Repository (MDR [169]) supports the core features of the Fujaba Association Framework (bidirectionality and integrity) as well as the three model transformation requirements related to this thesis. Therefore, this thesis relies on a new code generator that maps Story Diagrams to Java code that integrates directly with MDR, or any other JMI based repository.

5.2.2 Solution: Translating Story Diagrams to JMI Code

This section describes the overall architecture of the *Model driven, Template based, Model Transformer (MoTMoT)*, a tool that translates models conforming to the UML profile for Story Diagrams into Java code that satisfies all requirements outlined above. Interestingly, MoTMoT internally relies upon standards such as JMI as well. More specifically, it relies upon AndroMDA [29], an industrial strength UML-to-text transformation tool.

Subsection 5.2.2.1 discusses how transformation models are loaded, subsection 5.2.2.2 discusses how specific transformation elements are looked up in the generic UML input model, while subsection 5.2.2.3 refers to the definition of the generated code in the concrete syntax of the target language.

Figure 5.2: Definition of MoTMoT facade for metaclasses *Operation* and *Package*.

5.2.2.1 Loading Transformation Models

MoTMoT instructs AndroMDA to load UML XMI into a MDR repository. Since this repository is based on an official definition of the UML 1.4 metamodel, MoTMoT can consume transformation models that have been edited by any UML 1.4 compliant tool. In practice, MoTMoT has been tested with transformation models from Poseidon 2.2 [97], MagicDraw 9.0, and MagicDraw 9.5 [185].

5.2.2.2 Querying Transformation Elements

In order to facilitate the lookup of particular stereotype applications and tagged values, MoTMoT relies upon a set of facades. The AndroMDA tool chain is configured so that these facades are passed to the code generator instead of directly exposing the UML metaclass instances.

Figure 5.2 displays a fragment of the UML model from which the skeleton sources of these facades are generated. The upper classifiers that are displayed in white are part of an AndroMDA library. MoTMoT adds query operations to the *Operation* metaclass for facilitating the lookup of method body strings and related activity diagrams. The former is needed for operations that are modeled by plain Java code while the latter is needed for operations that are modeled by a story diagram.

The following code fragments illustrate what kind of queries are encapsulated within the facade implementations. The AndroMDA framework prescribes that query implementation methods are prefixed with “handle”. In the context of “handleGetMethodBody” for example, lines 56 to 59 are responsible for looking up the *Method* element associated to the *Operation* instance that is encapsulated by the facade. If such a *Method* element is found, its body is returned directly.

Some UML editors however do not enable one to instantiate such *Method* elements. Therefore, lines 66 to 70 “fail over” by looking up the *Comment* element associated with the underlying *Operation* instance. The code fragment from “handleGetTransformationFlow”

illustrates how the tagged between a method and a package is traversed (lines 88 to 91) to find the package holding the activity diagram that models the control flow of the transformation (lines 93 to 102).

```

20 package org.andromda.cartridges.motmot.metafacades;

...

38 public class MotMotOperationFacadeLogicImpl
39     extends MotMotOperationFacadeLogic
40     implements org.andromda.cartridges.motmot.metafacades.MotMotOperationFacade
41 {

...

55 public java.lang.String handleGetMethodBody() {
56     MotMotModelFacade model = (MotMotModelFacade) getModel();
57     Collection methods = model.returnCore().getASpecificationMethod().getMethod(
58         (Operation) metaObject
59     );
60     if (methods.size() > 0) {
61         Method method = (Method) methods.iterator().next();
62         assert( (method.getBody() != null) && (method.getBody().getBody() != null)
63             && (! method.getBody().getBody().equals(""))
64             ) : "Operation has no body (" + getName() + ")";
65         return method.getBody().getBody();
66     } else {
67         assert (((Operation) metaObject).getComment().size() > 0)
68             : "No body or comment found for operation (" + getName() + ")";
69         Comment comment = (Comment) ((Operation) metaObject
70             ).getComment().iterator().next();
71         assert (
72             ((comment.getBody() != null) && (! "".equals(comment.getBody())))
73             || ((comment.getName() != null) && (! "".equals(comment.getName())))
74             ) : "Operation body not specified (" + getName() + ")";
75         return ((comment.getName() != null) && comment.getName().equals("")) ?
76             comment.getBody() : comment.getName();
77     }
78 }

...

87 public ActivityGraphFacade handleGetTransformationFlow() {
88     MotMotPackageFacade transPkg =
89         (MotMotPackageFacade) findTaggedValue("motmot.transformation");
90     assert(transPkg != null) :
91         "Transformation flow package not found for method (" + getName() + ")";
92     // try ownedElements association first
93     Collection agf = new FilteredCollection(transPkg.getOwnedElements()) {
94         public boolean evaluate(Object object) {
95             return (object instanceof ActivityGraphFacade);
96         }
97     };
98     assert(agf.size() <= 1) :
99         "More than one activity graph found for method (" + getName() + ")";
100     if (agf.size() == 1) {
101         return (ActivityGraphFacade) agf.iterator().next();
102     }

...

115 }

...

144 }

```

5.2.2.3 Generation of Transformation Code

The actual transformation from the UML profile for Story Diagrams to JMI compliant Java code is defined as a set of dynamic content templates. These templates contain the “skeleton” of the generated code directly (i.e., without explicit output writing commands). Each template is parametrized with a specific element from the transformation model. From that variable parameter, all dynamic content can be accessed.

The following excerpt from MoTMoT’s *TransPrimitive.ftl* file shows the use of conditionals, assignments and list iteration in the context of the generation of the code for deleting an edge from the host graph. The expression *primPkg.destroyLinks* is supported by the *get-DestroyLinks* query operation shown in Figure 5.2. This helper will return all *Association* elements from the transformation model that are decorated with the «destroy» stereotype.

```

311 <@indent/>                                // remove links
312 <#list primPkg.destroyLinks as dLink>
313   <@prepareLinkVars link=dLink/>
314   <#if toOne>
315     <@indent/>                                ${sourceVar}.${setRoleName}(null);
316     <#elseif targetVar.equals(iteratorName)>
317     <@indent/>                                _motmot_${targetVar}_It.remove();
318     <#else/>
319     <@indent/>                                ${sourceVar}.${getRoleName}().remove(${targetVar});
320   </#if>
321 </#list>

```

The sample also illustrates the use of macros, by relying on *prepareLinkVars*. In this example, the macro initializes variables such as *toOne* and *targetVar* based on the *link* variable.

Figure 5.3: MoTMoT Architecture.

Figure 5.3 visualizes the interaction of components within the MoTMoT model transformation architecture for a refactoring example. The upper left rectangle represents the UML tool used to edit transformation models based on the UML profile for Story Diagrams. These transformation models can be stored in standard XMI and loaded into MoTMoT’s repository, shown within the upper right rectangle of Figure 5.3. The content of this repository is wrapped by MoTMoT’s facades and exposed to the dynamic content templates. These templates transform the visual transformation model into executable Java code.

That Java code can be executed to transform the models within a JMI based off-the-shelf (COTS [16]) repository or XMI models that have been loaded into MDR. Note that the transformation model contains a UML class diagram representation of the MOF based metamodel

Figure 5.4: Fragment of the input metamodel (OMG UML 1.5).

to which the input model of the transformation conforms. This part of the transformation model can thus be said to model the input metamodel. On the other hand, the behavioral part of the transformation model, i.e. the Story Diagrams, can be said to model the Java code that contains calls to the repository interfaces that manage the input model.

5.3 Example: Pull Up Method

This section applies MoTMoT to the example that has been used throughout the previous chapters. Figure 5.4 shows the fragment of the UML 1.5 metamodel that is relevant to this “Pull Up Method” refactoring.

The behavioral part of the transformation model is shown in Figure 5.5. As evaluated by the discussion in Section 5.1.3, Story Patterns are not visually embedded within states from the activity diagram. Instead, the reference between the different diagram types is maintained by means of the “*motmot.transprimitive*” tagged value discussed in Section 5.1.2.

Remark that this representation enables one to create transformation models that are self-documenting. Obviously, one should not claim that any modeler will attach meaningful names to all states from a transformation. However, the profile does stimulate this activity.

The transformation flow shown in Figure 5.5 (a) illustrates the use of “link states”: the “check precondition” activity is decorated with the **«link»** stereotype and is associated with exactly one “call action”, as shown by the MagicDraw pop-up window. This UML “call action” is defined by a call to the Story Diagram modeling the precondition. If this call returns success, the “perform transformation” activity is executed.

The behavior of this activity is modeled by the Story Pattern shown in Figure 5.5 (b). This Story Pattern looks almost identical to that from Figure 4.2, except for the explicit use of the **«bound»** stereotype. Obviously, a key difference is its standard compliance.

Figure 5.6 illustrates how iterative loops can be represented by showing the Story Diagram that models the precondition. The semantics of the Story Diagram is the same as that of Figure 4.3 from the previous chapter. The diagram also illustrates the use of the “*motmot.constraint*” tag. In this example, it is used to check whether all parameters from the method of the superclass and those from the method that is pulled up have the same type.

The transformation model presented in this section can be edited with the MagicDraw tool

Figure 5.5: Transformation model of the *Pull Up Method* refactoring.

Figure 5.6: Story Diagram modeling the precondition of “Pull Up Method”.

as well as with the Poseidon tool [253]. MoTMoT can load the XMI of this model and generate a complete plugin for Poseidon [97]. This tool has a JMI compliant UML 1.5 repository, such that the transformation can be executed directly (in-memory) on the repository interface. This eliminates an XMI serialization step which not only favors the performance of the resulting system but also ensures optimal preservation of layout information. Figure 5.7 displays the plugin pop-up window that appears when one right-clicks on a method.

As specified in the Story Diagram from Figure 5.5 (a), the method will only be pulled up if the precondition of the refactoring is met. In this example, the *Cabrio* class has a superclass *Car* and *Car* does not yet have an “accelerate” method returning *void*. Thus, “accelerate” will be moved from the *Cabrio* class to the *Car* class.

5.4 Related and Future Work

This section describes two Fujaba extensions that have adopted and/or extended some of MoT-MoT’s functionality. Subsection 5.4.1 indicates that the template based approach discussed in Section 5.2.2.3 has been adopted in the “official” branch of Fujaba too. Subsection 5.4.2 relates to the support for OCL.

Figure 5.7: Screenshot of the automatically generated UML CASE tool plugin.

5.4.1 Fujaba Codegen2

The original purpose of using dynamic content templates was to simplify Fujaba’s code generation architecture and thus facilitate support for other platforms than JMI based repositories. However, additional templates for new platforms like the Eclipse Modeling Framework (EMF [171]) have not yet been defined as MoTMoT was developed as a proof of concept only.

Interestingly, the more recent Fujaba *Codegen2* project has adopted the use of dynamic content templates for translating Story Diagrams to Java code too [140]. Templates for EMF and other frameworks are being defined at the time of writing.

The *Codegen2* project is designed such that optimizations within the Story Diagram code generation process can be modeled using Story Diagrams as well. The MoTMoT facades discussed in Section 5.2.2.2 also enable optimizations at the transformation model level but since the AndroMDA/MoTMoT facades are not designed with JMI in mind, MoTMoT has not yet been used to model its own code generator.

Instead, optimizations can be implemented directly as Java methods of the facades, or within the dynamic content templates, which is even less desirable. Chapter 8 illustrates how new language features can be added by means of higher order transformations. In principle, the same approach can be used for modularizing, and even optimizing, the code generation process for the “basic constructs” discussed in this chapter as well. However, we have focused our implementation effort on new language constructs instead of providing “yet another compiler” for already supported graph transformation constructs.

Finally, remark that the *Codegen2* project still relies upon the implicit and proprietary Story Diagram metamodel whose disadvantages have been discussed in Section 5.1.1. To align MoTMoT back with the main Fujaba development stream, one should define an explicit metamodel for Story Diagrams and then work in two directions. On the one hand, one should refactor and/or wrap the Fujaba repository to expose JMI (or EMF) compliant interfaces that conform to that explicit metamodel for Story Diagrams. On the other hand, one should define higher order transformations between the UML profile for Story Diagrams presented in this chapter and the explicit metamodel for Story Diagrams.

5.4.2 OCL Support

MoTMoT has been designed with OCL support at two levels. On the one hand, OCL is used to check the well-formedness of transformation models with respect to the UML profile for Story Diagrams. On the other hand, an OCL evaluator can easily be plugged into the generated Java code such that the transformation flow can be driven by OCL constraints at run-time.

As indicated in Section 5.1.3, fully standard compliant UML editors should be able to load profile well-formedness rules and check these automatically. Unfortunately, most of today's UML tools do not offer this functionality yet. Therefore, MoTMoT should check the well-formedness rules at the time transformation models are loaded into its repository. Unfortunately, no MDR related software supported the evaluation of OCL at the time MoTMoT was developed.

More recently, the YATL4MDR tool [64] did add the required infrastructure but hasn't been fully integrated with MoTMoT due to the immaturity of its syntactical and semantical analyzers. Still, the well-formedness of transformation models can be verified after they have been serialized to XMI and imported in a dedicated OCL constraint checker. Schippers has defined several OCL well-formedness rules for Story Diagrams and has evaluated them using the OCLE tool [43, 224].

The second level of OCL support is *within* transformation specifications. More specifically, the application conditions related to the "motmot.constraint" tagged value are evaluated through a pluggable interface. The current implementation evaluates Java constraints at run-time but another implementation could be added for supporting OCL constraints. OCL is more platform independent than Java as it abstracts from the differences between the JMI and EMF frameworks.

Again, the YATL4MDR tool has been investigated as a promising OCL evaluator but an actual implementation has been postponed due to the immaturity of that OCL evaluation library. In the meanwhile, Stölzel et al. have integrated the Dresden OCL Toolkit [149] with another Fujaba derivative [242]. The authors also propose semantical rules for restricting the set of variables that can be used within an embedded OCL constraint.

5.5 Summary and Outlook

The standardization of Story Diagrams presented in this chapter solves two significant issues that restricted the applicability of Fujaba within an industrial MDA context.

First of all, the UML profile enables one to use the UML CASE tool one is most familiar with as a transformation modeling environment. Secondly, the employment of pluggable dynamic content templates enables the transition to new execution platforms. Within the context of this thesis, support for JMI based repositories has been elaborated, which enables the integration with off-the-shelf repositories. At the same time, XMI models conforming to any MOF metamodel can be loaded and stored by relying on the mature MDR platform.

The proposed standard architecture is supported by the "Model driven, Template based, Model Transformer" (MoTMoT) tool. This tool has been used to illustrate that the techniques from the previous chapter can be decoupled from the Fujaba toolsuite. More specifically, the chapter demonstrated how the Pull Up Method refactoring can be modeled in MagicDraw and deployed as a completely generated plugin onto Poseidon. The upcoming chapters investigate whether these techniques are applicable to other transformation types too.

Part II

Synthesis

The previous chapter introduced a new, model-driven, approach to transformation development. From a modeling perspective, one can now rely on a combination of the semantical power of the Story Diagrams formalism with the exchangeability of standard UML syntax. Moreover, from an execution perspective, one can now integrate with MDA compliant model repositories. Finally, from an evolution perspective, one can more easily migrate the supportive code generator to new modeling platforms.

To illustrate these concepts in practice, the two previous chapters discussed the model-driven development of a refactoring plugin. The upcoming chapters validate and extend the proposed techniques in the context of another type of transformations. More specifically, the following three chapters relate to model *synthesis*.

Chapter 6 investigates in more detail to what extent model-driven techniques support the development of transformation models that are independent of the target modeling platform. Thus, instead of focusing on the “exchangeability” of transformation models in general, the chapter relates to the “platform independence” of specific transformation models. Recall that the exchangeability of a transformation model relates to the ability to edit such a model with different tools. In contrast, the platform independence of a transformation model relates to the ability to consume input models and/or write output models from/to a variety of modeling tools.

Chapter 6 illustrates how the platform Independence of a refinement can be supported by complementing it with an auxiliary normalization. It should be noted that the auxiliary normalization is not specific to the particular refinement. Nevertheless, the normalization is specific to the input metamodel of that transformation.

As a second contribution, Chapter 6 illustrates how the complexity of large rewrite rules can be decomposed by means of “transformation views”. In summary, the proposed technique enables one to model different aspects of a Story Pattern in different views. Overlapping view fragments are kept consistent automatically by the modeling tool.

Chapter 7 investigates how the copying of subgraphs within (or across) models can be modeled. The chapter illustrates that while such operations could already be programmed with the existing constructs for controlled graph transformation, a new language construct significantly improves the understandability of the transformation models. After discussing the semantics of the so-called “copy operator” in natural language, the new graph transformation construct is integrated in the UML profile for Story Diagrams.

Finally, Chapter 8 formalizes the semantics of the copy operator denotationally. More specifically, the chapter presents a set of mapping rules between the Story Diagram formalism with the copy operator and the conventional variant of the formalism, that is: Story Diagrams without the copy operator. Moreover, the chapter discusses how these mapping rules have been realized by means of Story Diagrams.

Instead of discussing that second-order transformation in the detailed, introductory, style used throughout the previous chapters, the chapter points to more design related trade-offs and process-related techniques. For example, it turns out that, thanks to transformation views, large rewrite rules may even have a positive impact on the understandability. More specifically, one can generate maintenance task specific views, which relates to recent work on *intentional views* [157, 156].

Chapter 8 also reflects on the fundamental differences between first-order and higher-order transformations. It turns out that more research is needed on distributed, multi-version development of transformations that support the execution of other transformations.

Chapter 6

UML2CSP: a Platform Independent Transformation Model

This chapter is based on a submission to the AGTiVE tool contest in 2007 [265] and is the basis for a paper that will be published in the proceedings of the MoVaH workshop in 2008 [201]. A summary of this chapter will also be published in the AGTiVE proceedings [271].

In an industrial context, a transformation should be able to consume input models from a variety of commercial tools. Unfortunately, such tools tend to store models in slightly different ways and do not fully comply to standards. Obviously, it is undesirable to build another version of a transformation for each specific modeling tool. Instead, there is a need for additional transformation techniques that enable the development of transformations in a platform-independent manner.

This chapter illustrates how the UML profile for Story Diagrams enables one to define a transformation from visual process models (UML activity diagrams) to low-level algebraic (CSP) programs that support formal verification. In turn, the platform independent, human-readable, and visual transformation model is translated into code that integrates with an activity diagram tool that slightly deviates from the UML 2.0 standard.

The transformation is designed as a chain that consists of a normalization and a synthesis step. The normalization can be considered as a “preprocessor” that removes all violations to the UML 2.0 standard. Consequently, the actual translation from UML to CSP can be modeled without taking these violations into account and remains independent of the concrete UML tool that is used to create activity diagrams. The normalization is realized as an in-place, endogenous transformation, while the translation is realized as an out-place, exogenous transformation of a UML sub-model to a first-class CSP model.

6.1 Context: Model Transformation Tool Contest

In order to facilitate a comparison of the expressiveness, usability and performance of graph transformation tools, three case studies have been proposed in the context of the third in-

ternational workshop on *Applications of Graph Transformation with Industrial Relevance* (AGTiVE [246]). In particular, a mapping from UML activity diagrams to Communicating Sequential Processes (i.e., the *UML-to-CSP* transformation) was proposed as a “*model transformation tool contest*”.

Chapter 5 introduced MoTMoT as a tool prototype that was built to illustrate how several model transformation problems of the Fujaba tool could be solved. As indicated in Section 5.4, several features make MoTMoT unique until today. Therefore, the AGTiVE “tool contest” was an excellent opportunity to present these features to potential MoTMoT users and to the developers of other graph transformation tools. Moreover, the contest was an opportunity to promote new transformation modeling techniques that can be applied by users of other tools too.

This chapter describes the MoTMoT solution to the UML-to-CSP case study and is structured as follows: Section 6.2 briefly introduces the reader to the description of the benchmark in general and more extensively discusses in what industrial scenarios the features of MoTMoT are most relevant. Section 6.3 presents the Story Diagrams that model the MoTMoT *UML2CSP* solution. Since the chapter is still oriented to readers that have very little familiarity with Story Diagrams or controlled graph transformation in general, the discussion of the diagrams is very elaborate. Section 6.4 summarizes the lessons one can learn from this case study. Finally, the chapter concludes by evaluating the strengths and weaknesses of the proposed approach.

6.2 UML-to-CSP: General and Specific Requirements

The UML-to-CSP transformation that is discussed in this chapter is based on the problem description of Bisztray et al. [26]. Instead of repeating the relevance of the problem and the sample input and output model, subsections 6.2.1 and 6.2.2 focus on the challenges that were identified by Bisztray et al. Subsection 6.2.3 describes the additional challenges that are tackled by this chapter. Moreover, an example of more complex input and output models is presented.

6.2.1 General Requirements of the Case Study

The problem description refers to four challenges that need to be tackled by any tool-supported solution [26]: first of all, the tool should support the notion of metamodels (i.e., type graphs). Secondly, the rewriting language should support the updating of node attribute values. Thirdly, it should be possible to define rule application conditions in terms of such attribute values. Finally, rewrite rules need to be embeddable in a control flow. From Chapter 4 and 5 it may be evident that these are only the basic features of Story Diagrams that are not considered as our research contribution.

6.2.2 Optional Requirements of the Case Study

As an optional requirement, the problem description refers to verification support. Due to the expressiveness of Story Diagrams (e.g. support for recursion), MoTMoT is unable to decide in general whether or not a particular transformation will always terminate. However, MoTMoT does support several syntactical and semantical checks.

First of all, the underlying java compiler reports any node type ambiguities or conflicts that are left in a transformation model. These checks revealed the differences between the specification of the simplified UML 2 metamodel given by Bisztray et al. and that of the commercial modeling tool used to test the discussed MoTMoT transformation. More specifically, the unqualified type names from the simplified metamodel are highly ambiguous (due to the presence of the given metaclasses in more than one package of the UML 2 metamodel).

Additionally, the UML 2 metamodel from the case study assignment subtly differs from the standard OMG UML 2.0 metamodel. For example, the simplified metamodel defines the name attribute only in the *ActivityEdge* and *Action* classes whereas this name attribute is actually inherited from a shared superclass in the standard OMG UML 2.0 metamodel.

MoTMoT also supports semantical checks on the control structure of a transformation. For example, it checks whether a sequence of states within a loop body ends in the state where the nodes over which the loop is iterating, are updated.

6.2.3 Additional Challenges concerning Industrial Relevance

The MoTMoT solution presented in this chapter tackles two additional challenges. Subsection 6.2.3.1 presents the first challenge: the case study solution should be a platform independent transformation that is executable on non-standard tools. The second challenge is presented in subsection 6.2.3.2: the transformation solution should realize the mapping from UML to CSP in a robust manner. Even activity diagrams that are not of the canonical form that is presented in the problem description from Bisztray should be transformed correctly to corresponding CSP expressions.

6.2.3.1 Platform Independent Transformation Modeling

In an industrial context, one needs to reconcile the following two conflicting concerns:

- On the one hand, it is desirable to model the transformation independently of the particular tool that is used to produce input UML models.
- On the other hand, the generated transformation code should correctly transform UML models produced by a concrete, industrial, tool.

The obvious solution to the first concern is to model the mapping from UML to CSP in terms of the standard UML metamodel. However, that is in conflict with the second concern since industrial tools tend to store models in a non-standard manner.

To encounter these issues in practice, we relied on MagicDraw 10.0, an industrial UML 2.0 tool that deviates slightly from the OMG standard. The vendor of MagicDraw does provide libraries to load the XMI 2.1 files produced by MagicDraw 10.0 into a MOF compliant repository [186]. However, the content of that repository is not completely OMG UML 2.0 compliant. More specifically, the following concrete challenges need to be overcome:

1. *MergeNode* elements are represented as *JoinNode* elements whereas *JoinNode* elements are represented as *ForkNode* elements. This violates some UML 2.0 well-formedness rules (WFRs) but is a given fact that needs to be dealt with when integrating with industrial tools.

2. The inheritance links induced by the MOF 2.0 package merges in the UML 2.0 meta-model definition are realized by MagicDraw-specific metaclasses.

Figure 6.1: Metamodel imposed by the industrial UML tool used to produce input models.

Figure 6.1 shows the fragment of the MagicDraw 10.0 metamodel that highlights the concrete integration problem following from the proprietary way in which the inheritance links induced by package merges are realized (challenge 2): instead of providing a specialization link from *uml2.basicactions*'s *Action* class to *uml2.basicactivities*'s *ActivityNode* class, the merge of the “actions” hierarchy with that of the “activities” hierarchy is realized in the MagicDraw-specific *CallBehaviorAction* class from *uml2.mdbasicactions*.

Note that all classes in packages with the “md” prefix are MagicDraw-specific, which is emphasized in red in Figure 6.1. The next section will illustrate that in order to model the transformation in a human oriented manner, one needs to reason about nodes that are both of type *Action* and of type *ActivityNode*. However, the transformation model cannot rely on the *CallBehaviorAction* class from *uml2.mdbasicactions* as this would make the model MagicDraw-specific.

In general, these two issues illustrate that the use of commercial, off-the-shelf, modeling tools leads to challenges that are not encountered when using “in-house” academic software only. The following sections also tackle these issues to illustrate the flexibility of MoTMoT.

6.2.3.2 Robustness of the Mapping

The problem description of the tool contest does not define strict compliance points [26]. Instead, the document prescribes the input and output metamodels, a (rather limited) set of mapping rules, which are illustrated by an example run of the expected kind of transformation. Without claiming completeness, this section points out that the solution discussed in this chapter takes some additional robustness requirements into account.

Figure 6.2: Example UML 2.0 input activity diagram challenging the robustness of a solution.

Figure 6.2 illustrates what kind of models should be handled by a more robust version of the *UML-to-CSP* transformation too. The decision element shown at the left side of the diagram illustrates that, in some cases, no outgoing transition has an *ELSE* guard (which is an implicit assumption in the default mapping rules). This challenge has also been identified by Rencis [210].

Transformations should be able to handle input process models in general, even when the semantics of such models is questionable. In fact, when particular process descriptions are considered to be senseless, it is up to CSP constraints to forbid such configurations. Therefore, transformations should, for example, be able to handle direct transitions from fork to join nodes. The pair of fork and join elements shown in the middle of the diagram exercise this robustness requirement. Similarly, the fork element shown at the right of the diagram expresses that the *K* and *L* activities should be executed in parallel. As a potential modeling error, these concurrent activities are not synchronized before the end of the process.

From a more syntactical perspective, the “fork” element at the right of the diagram violates the UML well-formedness rules. More specifically, the element also synchronizes the preceding *H*, *I* and *J* activities. A transformation that only consumes UML 2.0 standard compliant inputs will not be able to handle such configurations. Still, the input model shown on Figure 6.2 has been produced with an industrial tool that claims compliance to that standard.

As a final robustness test, the sample input model contains a decision node with only one outgoing transition. Although such elements do not model an actual decision, they may be specified in practice.

The following CSP code corresponds to the UML model shown on Figure 6.2. It should

give the reader a feeling of the kind of expressions that needs to be generated by the transformation.

<pre> 1 start2a = A → a2decision 2 a2decision = (3 decision2J < grd_dec2J > (4 decision2I < grd_dec2I > (5 decision2H < grd_dec2H > (6 decision2merge < grd_decision2merge > (7 decision2fork < grd_dec2fork > 8 decision2d 9) 10) 11) 12) 13) 14 decision2fork = (fork2join fork2b) 15 c2end = SKIP 16 merge2c = C → c2end 17 fork2b = B → B2join 18 d2e = E → e2merge 19 d2f = F → f2merge 20 decision2d = (21 d2f < guardOn_d2f > (22 d2e < guardOn_d2e > SKIP 23) 24) </pre>	<pre> 25 c2singleDecision = SKIP 26 decision2H = H → H2MNjoin 27 decision2I = I → I2MNjoin 28 decision2J = J → J2MNjoin 29 MNjoin2K = K → K2merge 30 MNjoin2L = L → L2merge 31 TrMNjoin_to_MNjoin = (32 MNjoin2L MNjoin2K 33) 34 decision2merge = merge2c 35 join2merge = merge2c 36 f2merge = merge2c 37 e2merge = merge2c 38 K2merge = merge2c 39 L2merge = merge2c 40 fork2join = processJoin → join2merge 41 B2join = processJoin → SKIP 42 H2MNjoin = processJoin → 43 TrMNjoin_to_MNjoin 44 I2MNjoin = processJoin → SKIP 45 J2MNjoin = processJoin → SKIP </pre>
--	---

6.3 MoTMoT: Transformation Model for UML-to-CSP

This section describes the MoTMoT artifacts that have been developed as a solution to the UML-to-CSP case study. Subsection 6.3.1 discusses the architecture of the solution by modeling the interaction of the generated MoTMoT transformation class with its environment. Subsection 6.3.2 describes the structure of the transformation class along with its input and output metamodels while subsection 6.3.3 describes the behavior of the methods that realize the actual mapping from UML elements to CSP constructs. Finally, subsection 6.3.4 explains what techniques have been applied to ensure the transformation produces correct output even when its input does not exactly conform to the standard UML 2 metamodel.

6.3.1 Architecture

This subsection introduces the reader to the architecture of the solution by describing which code artifacts are generated and which ones are written by hand. Moreover, the role of the different tools is modeled for a sample run of the transformation.

Figure 6.3 displays how a UML 1 tool can be used to edit transformation models conforming to the UML profile for Story Diagrams. MoTMoT applies a set of AndroMDA templates to generate a Java implementation from such a transformation model. In the example given, the *UML2CSPImpl.java* file displayed at the top of the diagram, represents such a generated implementation file.

The UML2CSP transformation class is instantiated from a JUnit test, called *UML2CSP-Test.java*. The test reads the input XMI file, produced by a UML 2 tool, into a MOF repository and loads the activity diagram that needs to be transformed by the *UML2CSP* instance. After executing the “transform” method of the transformation instance, the resulting *CspContainer*

Figure 6.3: Architectural model of the Case Study Solution

instance is serialized to XMI again. Finally, an XSL template transforms the abstract syntax instances from the XMI file into expressions in actual CSP syntax.

All case study artifacts are available in the “*samples/uml2csp*” directory of the MoTMoT distribution [178]. A “standalone” version of the sample is available for download too [263]. This version automatically deploys and runs all the required software on a typical shell-based system.

In “model.xml”, several sample activity diagrams are defined in MagicDraw 10 (i.e., UML 2) format. The “ActiveSample” diagram is a copy of the sample from the benchmark description. Other activity diagrams have been added to illustrate the generality and robustness of the solution that is presented in this chapter. The file called “metamodel.xml” contains the MOF 1 metamodel defining UML 2. The file called “transformation.xml” has been constructed with MagicDraw 9 and contains all UML 1 based Story Diagrams presented in the following sections.

6.3.2 Structure: Related Elements and Mapping Responsibilities

This subsection describes how the structure of the transformation model relates to that of the input and output models. The structure of the input metamodel has been discussed in the previous section. The output metamodel is structurally identical to the CSP metamodel from [26] but is made more precise giving explicit names to all association ends (which is required to resolve an ambiguity in the *ProcessExpression* class, due to the two incoming compositions starting from the *BinaryOperator* class). Figure 6.4 presents the MOF compliant metamodel for the structure of the UML2CSP transformation.

As Figure 6.4 indicates, each *UML2CSP* instance should have exactly one reference to an

Figure 6.4: Structure of the UML2CSP transformation model.

Activity instance with name *in*. This reference represents the input UML 2 activity diagram that needs to be transformed into a *CspContainer*. Quite simply, this task can be accomplished by calling the only public method of the *UML2CSP* instance: “*transform*”. The behavior of this method is realized by a subclass called *UML2CSPImpl*.

As can be seen from the corresponding stereotypes, the six other *transform* methods in *UML2CSPImpl* realize a model transformation “rule”. Each such method has two parameters: the first parameter represents the input UML model element that needs to be transformed while the second parameter represents the *CspContainer* to which generated CSP assignment expressions need to be added. These transformation rules are strictly *out-place* since they create elements in the output model while leaving the input model unaffected. Moreover, the transformation rules are exogenous since the input and output elements are typed with classes from different metamodels.

Coming to the technical aspect of defining an abstract method that is overloaded by the six transformation methods, the second challenge following from the industrial constraints (discussed in Section 6.2.3) comes into play. More specifically, one would want to provide an abstract method *transform(inputElement, outputContainer)* where the first parameter is of type *ActivityNode* and the second parameter is of type *CspContainer*. However, due to the lack of a specialization link from *Action* to *ActivityNode*, the first parameter is typed with the more general *NamedElement* metaclass. This undesirable mismatch between the transformation writer’s intention and the required MoTMoT transformation model is a known limitation for which a general modeling solution has not yet been proposed. Instead, it is treated as an implementation aspect that is hidden from diagrams such as the one from Figure 6.3.2. A similar, yet orthogonal, issue is discussed in Section 6.3.5.

The *motmot.transformation* tagged value on each of the transformation methods indicates in which package one can find the Story Diagram modeling the behavior of the method under consideration. The two transformation methods whose name starts with “standardize” contribute to the robustness of *UML2CSPImpl*. More specifically, they normalize non-standard constructs from the input UML in-place transformations into standard UML constructs by means of in-place transformation steps.

Finally, Figure 6.4 indicates that *motmot.ispartofmodel* is true for *UML2CSPImpl*. Additionally, it indicates that this transformation class specializes the *InstanceHandler* interface.

This configures MoTMoT such that the objects of type *UML2CSP* can be stored as a model element in a repository too.

6.3.3 Behavior of the Transformation Methods

This subsection presents the story diagrams that model the behavior of the complete transformation. These story diagrams are presented in the same chronological order as the mapping rules in section 3 of the case study’s problem description [26].

6.3.3.1 Transform Action Node

The mapping of an UML Action to a CSP *ProcessAssignment* can essentially be realized in one step, that is modeled by the Story Pattern (i.e., “primitive graph transformation rule”) shown in Figure 6.5.

Figure 6.5: Story Pattern Transforming Action Nodes.

As introduced in Chapter 5, Story Patterns can be encoded in standard UML 1.x using stereotypes and tagged values on class diagrams. As stated in Section 5.1.2, classes and associations without a stereotype, or those with the `«destroy»` stereotype represent nodes or edges that need to be matched in the host graph. They are thus part of what is commonly known as the left-hand side of the rewrite rule. Elements with a `«destroy»` stereotype need to be removed when a match is found. They thus correspond to elements that are part of the left-hand side of the rewrite rule without being part of the right-hand side. Conversely, elements with the `«create»` stereotype should be created when a match is found. They thus correspond to elements that are part of the right-hand side of the rewrite rule without being part of the left-hand side. Nodes marked as `«alias»` are direct aliases for bound nodes. Aliased nodes can have a more generic type than the nodes they are rebinding.

Node types are specified using the *motmot.metatype* tagged value, node attributes are specified as class attributes and assignments to attribute values are specified as initial values (i.e., using the “=” character). Note that node types are fully qualified within transformation models although the diagrams in this chapter only show unqualified type names as values for *motmot.metatype*.

Edges are typed by the pair of association end names. These edge types can be expressed more concisely by using unidirectional association links. This technique not only saves space on Story Diagrams, it also influences the matching process: at rule application time, the link will be traversed in the direction indicated by the Story Pattern.

Throughout this chapter, we apply the following stylistic conventions to improve the readability of Story Patterns: nodes that are bound are colored light-red, nodes that need to be matched are colored gray, nodes (and links) that need to be removed are colored red and nodes (and links) that need to be created are colored green. Moreover, UML dependency links (dashed arrows) are used to indicate that the source node of the dependency link is generated from the target node of the dependency link. Note that these annotations are applied for documentation purposes only and thus have no effect on the MoTMoT code generator.

Coming back to Figure 6.5, one can now read this Story Pattern as follows: if the “input” *Action* node has both an input transition *in_edge* as well as an output transition *out_edge*, generate a corresponding CSP assignment by creating a process with the same name as the *in_edge* at the left-hand side of the assignment operator (i.e., as the *processIdentifier* of the assignment) and a prefix expression at the right-hand side of the assignment operator. The event of the prefix expression gets the name of the input *Action* node while the *Process* node at the right-hand side of the prefix expression’s arrow operator gets the name of the output transition (i.e., *out_edge.getName()*). Note that the *actionAsActivityNode* and *out* nodes are already bound at rule application time since they are passed as arguments to the transform method: the *out* node corresponds to the second parameter of the transform method, while the *actionAsActivity* node is an alias for this method’s first parameter. Because the *-owner—processAssignments-* edge carries the «create» stereotype, the generated CSP assignment expression will be added to the output CSP container *out*. Finally, observe that a complete example of the «alias» syntax will be presented in the context of the transformation of fork nodes.

6.3.3.2 Transform Initial Node

The transformation of initial nodes does not require a dedicated rule since the expected process assignment is already generated by the other mapping rules. More specifically, all other mapping rules generate assignments to processes whose name is based on that of an incoming transition. If such a transition starts from the initial node, that node will be transformed appropriately too.

6.3.3.3 Transform Final Node

The transformation of a final node requires the creation of a simple assignment from the final node. As Figure 6.6 shows, the process at the left of the assignment operator gets the name of the input transition of the final node. Syntactically, this name assignment is realized by means of the attribute assignment on the *leftProcess* node. Using the same syntactical construct, the process at the right of the assignment operator gets the name “SKIP”. The Story Pattern has a

Figure 6.6: Story Pattern Transforming Final Nodes.

very small application condition: it only requires the final node to have an incoming transition *in_edge*. The Story Diagram that embeds the above Story Pattern in a trivial control flow is not shown.

6.3.3.4 Transform Decision Node

The transformation for decision nodes should generate a CSP assignment expression with at the right-hand side a tree of conditional expressions. An example of such a mapping is shown in Figure 6.7, which is taken from the case study description [26].

Figure 6.7: Example Mapping of a Decision Node.

As can be generalized from this example, the outgoing transition holding a guard called “else” should be mapped to a process that is nested most deeply within the tree of conditional expressions at the right-hand side of the CSP assignment. When providing rewrite rules for all elements involved in the input UML expression (i.e., the input transition, the output transition holding the “else” guard, and finally all other transitions) one requires a means to embed these primitive graph transformation rules in a control flow. In the UML profile for story diagrams, this is realized by decorating an activity diagram with stereotypes that specialize the generic state elements of standard UML and with tagged values that link a state to a Story Pattern or that capture constraints that cannot be expressed elegantly using Story Patterns.

Figure 6.8 shows the Story Diagram that controls the flow between the different Story Patterns for mapping a decision node to a compound CSP expression. The given story diagram realizes a purely imperative approach that starts with generating the most deeply nested

expression (“D” in the example given above), keeps track of the most recently generated expression and iteratively generates conditional expressions that have a process for another outgoing transition (*B* or *C* in the example given above) at the left-hand side and the previously generated expression at the right-hand side. As long as outgoing transitions can be found that have not been transformed before, new conditional expressions are generated. Afterwards, the outermost conditional is used as the right-hand side of an assignment expression that is generated for the incoming transition of the decision node.

Figure 6.8: Story Diagram Transforming Decision Nodes.

The Story Diagram in Figure 6.8 shows that at first, two auxiliary variables are initialized. From the previous paragraph, one may understand that these two variables are needed to control the rewrite rules:

- one variable keeps track of the most recently generated CSP expression for embedding that expression within the iteratively constructed expression tree,
- another variable keeps track of what transitions have already been transformed.

After initializing these variables in the first state (using a rewrite rule with an empty left-hand side and two new nodes for the helper variables at the right-hand side), the *ElseTransition2Process* Story Pattern creates a process for the transition with a guard named “else”. This Story Pattern, shown in Figure 6.9, contains three bound nodes: the *decision* node is bound, as it represents the first argument of the transformation method, and the *handledEdges* and *lastlyGenerated* nodes represent the two helper variables that have been created in the first Story Pattern of the flow for transforming decision nodes. The types of these nodes are *EdgeContainer* and *ExpressionPointer* respectively. These types are defined in the structural transformation model of which a fragment was shown in Figure 6.4.

As the application condition of the *ElseTransition2Process* Story Pattern, consider the *out_edge* and *guard nodes* (i.e., those nodes that have no stereotype and that are colored in gray). These nodes ensure the rule only fires when the input decision node has an outgoing

Figure 6.9: Story Pattern named *ElseTransition2Process* generates a CSP process expression for the outgoing “ELSE” transition.

edge with a guard named “else”. The constraint of the guard name is expressed as an attribute value condition on the guard node. If its application condition evaluates to false, the *ElseTransition2Process* is not fired. On the control flow level (i.e., at the level of the Story Diagram shown in Figure 6.8), this leads to the firing of the *«failure»* transition between the *ElseTransition2Process* state and the *NoElse2SKIP* state. Consequently, the *NoElse2SKIP* rule is evaluated. The Story Pattern corresponding to this rule is shown in Figure 6.10.

Figure 6.10: *NoElse2SKIP*: Story Pattern for mapping a decision node without an outgoing “ELSE” transition to a CSP expression named “SKIP”.

The *NoElse2SKIP* rule does not require the presence of an “ELSE” guard. In fact, it does not even require the presence of an outgoing transition. This implies that the transformation for decision nodes should produce correct output in the case no output transitions are present.

The *NoElse2SKIP* rule generates a *Process* node with name “SKIP” and directs the *lastlyGenerated* expression pointer to that CSP *Expression* node, so that the subsequent *OutgoingTransition2ConditionalExpression* rule can use the generated SKIP expression as the right-hand side of a conditional expression. Note that in case no outgoing transitions are present, the transformation will continue directly to the *IncomingTransition2OutputAssignmentExpression* rule where the most recently generated expression, which is SKIP in this case, is used as the right-hand side of the generated CSP assignment expression.

An assignment to SKIP is a reasonable mapping for the exceptional case of a decision node without outgoing transitions, the solution presented here can be considered to be robust.

Since the *OutgoingTransition2ConditionalExpression* Story Pattern involves the creation of quite a number of nodes and edges, we decompose it into two complementary *transformation views*: while Figure 6.11 shows a diagram highlighting the core mapping responsibilities

Figure 6.11: View on *OutgoingTransition2ConditionalExpression* Story Pattern from the perspective of its core mapping responsibilities.

of the rule, Figure 6.12 shows a diagram focusing on rewrite nodes and edges that control the termination of the *OutgoingTransition2ConditionalExpression* rule and that connect expressions generated by different rules.

Figure 6.12: Transformation view on *OutgoingTransition2ConditionalExpression* Story Pattern from the perspective of rule interaction elements.

The mapping behavior shown in Figure 6.11 consists of generating a conditional expression for the given decision node. As visualized by a dotted edge, the expression of this conditional expression is based on the guard of the decision node's output transition. Another dotted edge visualizes that the name of this output transition is used to generate the process at the left-hand side of the conditional expression. The right-hand side of the conditional refers to the conditional that is the top of the expression tree generated by previous rewrite rules.

On Figure 6.12, one can observe that the *topConditional* node representing this top expression should be matched by navigating the link from the *lastlyGenerated* pointer. Note that this link is redirected once the new conditional expression is generated. Also note that the *out_edge2* node is added to the auxiliary *handledEdges* container. The constraint on the Story Diagram state referring to the *OutgoingTransition2ConditionalExpression* pattern (see Figure 6.8) ensures that an output transition is not transformed more than once into a CSP conditional expression:

```
!handledEdges.getEdges().contains(out_edge2)
```

This state constraint can also be modeled elegantly by a Negative Application Condition (NAC) but this syntactical construct is not (yet) supported by MoTMoT.

Figure 6.13: *IncomingTransition2OutputAssignmentExpression*: Story Pattern mapping the incoming transition of the decision node to the left-hand side of the assignment.

The final Story Pattern controlled by the Story Diagram for transforming decision nodes ensures that the conditional (or SKIP) expression, generated by the other Story Patterns of the decision node transformation, is embedded in the assignment expression that is added to the global sequence of CSP expressions. More specifically, as Figure 6.13 shows, the expression tree generated by previously executed transformation rules is looked up by matching the *topProcess* node from the already bound *lastlyGenerated* node. Once found, it is connected to the newly generated *assignment* node by means of a new *process* link. In the CSP metamodel from the case study, this link represents the right-hand side of the assignment. On the left-hand side of the assignment, a process is generated from the input edge of the decision node.

Note that the *in* node is an alias of the input *decision* node that is bound as a parameter of the transformation method. This alias is initialized in the transformation's first state and is provided to enable one to reuse the *IncomingTransition2OutputAssignmentExpression* Story Pattern across different Story Diagrams, as will be illustrated when discussing the transformation of fork nodes.

6.3.3.5 Transform Merge Node

One Story Pattern is sufficient to model the behavior of the transformation from merge nodes to the corresponding CSP assignments. Figure 6.14 shows this Story Pattern, which should be executed as long as matches can be found. Multiple matches of the primitive rewrite rule originate from the *in_edge* node that is iteratively mapped to all transitions that enter the given merge node. As visualized by the dotted edges, the process on the left-hand side of the assignment expression is based on the input transition while the right-hand side is based on the transition that leaves the given merge node.

The Story Diagram modeling the flow in which the above Story Pattern is embedded, is not shown due to its simplicity: it consists of only one state that is decorated with the `<<loop>>`

Figure 6.14: Story Diagram for transforming Merge Nodes.

stereotype.

6.3.3.6 Transform Fork Node

Figure 6.15: Story Diagram modeling the control flow of the transformation for Fork Nodes.

The control flow of the transformation for fork nodes strongly resembles that of the transformation for decision nodes. The transformation for fork nodes is simpler than that for decision nodes since outgoing transitions have no features that require a treatment different from that of other outgoing transitions. In contrast, recall that transitions leaving a decision node require a special treatment in the case they have a guard labeled “else”. Figure 6.15 models that in the second state of the fork node transformation, an output transition is selected at random and transformed to a CSP process. Afterwards, the transformation transforms all other

outgoing transitions into CSP concurrency expressions. Finally, the most recently generated CSP expression is used as the right-hand side of a newly created CSP assignment expression.

Figure 6.16: Story Diagram for initializing auxiliary nodes that support the control flow.

Note that the Story Pattern for the first state binds the same auxiliary nodes as discussed in the context of the decision node transformation discussed before. Moreover, it binds the *in* node as an alias of the *fork* node, in order to reuse a complete Story Pattern in the fourth state of the transformation. Figure 6.16 illustrates how the UML profile for Story Diagrams supports such node aliasing (including casting) by means of the `<<alias>>` and `<<rebind>>` stereotypes.

Figure 6.17: *OutEdge2Process*: Transformation of a randomly selected output transition.

Figure 6.17 models how in the second state, the output transition is selected at random: unlike the corresponding Story Pattern from the decision node transformation (i.e., *ElseTransition2Process*, displayed in Figure 6.9), the unbound *out_edge* node is not constrained by an association with a specific guard node. Apart from that, the meaning of the *OutEdge2Process* pattern is similar to that of the *ElseTransition2Process* pattern.

Apart from the type of the topmost expression and the name of the input node, the *OutgoingTransition2ConcurrencyExpression* pattern is identical to the *OutgoingTransition2ConditionalExpression* that has been presented by the two views shown in Figure 6.11 and 6.12. Figure 6.18 shows the complete pattern on one diagram.

As indicated above, the fourth state, called *IncomingTransition2OutputAssignmentExpression*, is reused from the decision node transformation. By creating an alias from the *in* node to the *fork* node in the first state of this transformation (modeled by the pattern from Figure 6.16), and by creating a similar alias from the *in* node to the *decision* node in the first state of the decision node transformation (discussed in the context of Figure 6.8), the name and type differences within the two client Story Diagrams have been encapsulated properly.

Figure 6.18: *OutgoingTransition2ConcurrencyExpression*: Story Pattern transforming an output transition of a fork node into a CSP concurrency expression.

6.3.3.7 Transform Join Node

Figure 6.19: Story Diagram of the transformation for Join Nodes.

According to Bisztray et al., the mapping of join nodes involves the most complex transformation. However, from a Story Diagram perspective, the control flow of the decision and merge node transformations is more complex than the one of the join node transformation discussed in this section. In fact, the following Story Diagram is only a minor variation on that for transforming merge nodes.

The Story Diagram for the merge node transformation consists of only one state that is decorated with the loop stereotype. In this state, an assignment is created for an incoming transition. Figure 6.19 shows that the Story Diagram for join nodes has only one additional state preceding this loop pattern.

Figure 6.20 shows the Story Pattern for the first state of the transformation. In this state, a transition that enters the input *join* node is selected at random. Once more, a *handledEdges* node is used to keep track of the transitions that have been transformed so far. Join nodes are supposed to have only one output transition. In the rule discussed here, the *out_edge* node is bound to the transition that leaves the input *join* node. When the random input transition and the unique output transition have been found, a newly created CSP expression is added to

Figure 6.20: *handle1randominput*: Story Pattern transforming one input transition to a special CSP assignment.

the output CSP container *out*. The left-hand side of the assignment refers to a process whose name is equal to that of the input transition. The right-hand side refers to a prefix expression consisting of a process whose name is equal to that of the output transition and an event named “processJoin”.

As Figure 6.21 shows, the transformation of the other input transitions is modeled by a very similar story pattern. Only one rewrite node differs conceptually from the previously discussed Story Pattern: the *name* and *subscript* values of the process created at the right-hand side of the prefix expression differ from the values assigned in the previous Story Pattern. This follows from the case study’s mapping requirements that require a subscripted SKIP statement to be generated for all blocking input transitions. The dotted edges emphasize that in the latter Story Pattern, both processes from the generated CSP assignment are based on the name of the input transition. On the other hand, the dotted edges in Figure 6.20 are intended to help the reader in verifying that the target process in the right-hand side of the assignment is generated from the output transition instead.

6.3.3.8 Transform IN to OUT

The Story Diagrams discussed in the previous sections are all based on the overloading of a generic transform method whose first parameter represents the input UML element to be transformed and whose second parameter represents the output CSP container to which the generated CSP expressions need to be added. This section shows how a simple Story Pattern can ensure that this primitive method is executed for all elements of a particular activity diagram.

The Story Diagram shown in Figure 6.22 models the behavior of the *_sdm_transform* (for “story driven modeling” transform) method. The generated implementation of this method

Figure 6.21: Story Pattern for transforming all other input transitions.

Figure 6.22: Top-Level Story Diagram: Transform IN to OUT.

realizes the behavior that is expected from the top-level *transform* method discussed in Section 6.3. As stated, it does not require any parameters in order to generate a *CspContainer* corresponding to the activity diagram referenced by the *in* link of the *UML2CSP* instance. The MoTMoT JMI code generated for the *_sdm_transform* method is not generated into the body of the *transform* method directly since it needs to be surrounded by some exception-handling code in order to satisfy the exception-free signature of the *transform* method.

Figure 6.22 illustrates how swimlanes can be used to separate the different concerns of a model transformation: the “Model Management” lane contains the states related to the actual mapping while the “User Feedback” lane holds some code states that produce configurable debugging output. Within the left swimlane, the first state creates an empty instance of a *CspContainer*. The following states will add CSP assignments to this container. The second state iterates over all nodes contained within the input activity diagram. Note that the swimlanes have no semantical effect on the code generator. Therefore, their layout is not constrained and

can be optimized in terms of documentation quality.

Figure 6.23: The set of source elements consists of all nodes n from the input activity diagram.

As Figure 6.23 illustrates, only nodes that are part of the input activity diagram are visited. Although this is accomplished in a trivial manner, such transformation scoping is not easily supported by strict graph grammar-based transformation approaches. The final state of the model management lane contains a method call to the overloaded *transform(in, out)* methods that are modeled by the Story Diagrams discussed in the previous sections.

6.3.4 In-Place Transformation removing non-standard constructs

This section illustrates that MoTMoT not only supports out-place transformations from one or more input model(s) to one or more output model(s): transformations that update an input model in-place can be modeled using Story Diagrams too. Such in-place requirements enable one to model a solution that meets the additional requirements outlined in Section 6.2.3 without too much specification effort.

Normalizing UML elements is needed when industrial editors store UML diagrams in a non-standard way. The popular MagicDraw tool, for example, offers in its 10.0 version an editor that instantiates UML *Decision* nodes for representing both decision nodes as well as merge nodes. Similarly, it makes improper use of UML *Fork* nodes by using them to represent join nodes too. This chapter presents a solution to the UML2CSP case that handles non-standard fork nodes as well as non-standard decision nodes. The normalization of these non-standard constructs is very similar. Therefore, only the diagrams for the decision node normalization are included here.

Figure 6.24: Control flow for normalizing non-standard decision nodes.

The Story Diagram shown in Figure 6.24 models a normalization that is not specific to the MagicDraw 10.0 variant of the UML. More specifically, it transforms any decision node with more than one input transition into a standard counterpart. This involves two cases:

- either the decision node is used with the compound semantics of a decision node and a merge node, by having multiple input transitions as well as multiple output transitions (carrying guards),
- or it is used as a merge node exclusively, by having multiple input transitions but only one output transition.

In both cases, a new merge node element needs to be created and all input transitions of the decision node need to be moved to the new merge node.

In the first state of the transformation, violating decision nodes (i.e., those with more than one input transition) are detected and a corresponding merge node is created for such decision nodes.

(a) Detection of violating decision node and creation of corresponding merge node (b) Moving a transition to the new merge node

Figure 6.25: Story Patterns for the first two states of the normalization for decision nodes.

Figure 6.25 (a) shows the Story Pattern for this state. Intuitively, the two input transitions are displayed above the violating decision node. The `<<create>>` stereotypes ensure that a newly created merge node is added to the input activity diagram. Note that the violating decision node is not (yet) removed as it may need to be preserved in the input activity diagram.

Only in the state that is displayed at the bottom of Figure 6.24, the decision node can safely be removed. Remark that in contrast to the `<<create>>` and `<<destroy>>` constructs presented in Section 6.3.3, the ones presented in this section do change the input model. Therefore, the transformation is said to be *input-destructive* while that of the previous section is *input-preserving*.

As indicated by the `<<loop>>` and `<<each time>>` stereotypes, the path of states at the right of the Story Diagram shown in Figure 6.24 is entered for each violating decision node. The first node on this path is decorated with the `<<loop>>` stereotype. This state is responsible for moving all input transitions from the violating decision node to the newly created merge node. Since there is no `<<each time>>` transition leaving this state, the execution thread remains local to the Story Diagram shown in Figure 6.25 (b) as long as it can be matched.

The second and third state on the path at the right of the Story Diagram from Figure 6.24 handle the differences between the two decision node violation scenarios discussed above: the second state normalizes decision nodes that are used at the same time as a decision node and as a merge node, while the third state handles the case where a decision node is abusively used as a merge node only (as in MagicDraw 10.0).

Figure 6.26 shows the Story Pattern for the second state. The gray nodes that are part of its application condition evaluate whether the given decision node has an output transition

Figure 6.26: Story Pattern *prepending* a generated *Merge* node to *Decision* nodes that are used as decision *and* merge nodes at the same time.

with a guard. If such a transition is found, the rule concludes the decision node takes an actual decision and therefore should be preserved within the input activity diagram. If such a transition cannot be found, the rule's side effects are not executed and the Story Pattern for normalizing decision nodes triggers the outgoing `<<failure>>` transition. This transfers the transformation in the state where it should replace the decision node, that is obviously used as a merge node, by the generated UML Merge node.

As Figure 6.27 shows, the Story Pattern for this state deletes the violating decision node from the input activity diagram after redirecting its outgoing transition to the newly generated merge node.

Figure 6.27: Story Pattern *replacing* a violating decision node by a generated merge node.

6.3.5 Declarative Rule Scheduling

The UML2CSP case study raises an interesting challenge with regards to conceptual transformation modeling, as opposed to machine-oriented transformation programming. More specifically, the scheduling of the different *transform(activityNode, cspContainer)* rules presented in Section 6.3.3 should be as follows: transform each activity node *n* with the *transform* rule whose first argument matches the dynamic type of *n*. If there is no rule whose first argument can be matched with *n*, decide that *n* should not be transformed and proceed to the next matching iteration.

Since Story Diagrams do not provide any declarative rule selection operators (e.g., “unconstrained” matching, cfr. Figure 2.3 from the taxonomy), we investigated how the desired behavior could be realized using explicit method calls from a delegating rule that is matched for all activity nodes.

At this level, it should be taken into account that MoTMoT's low-level execution platform

is the Java 1.4 virtual machine. While the strong typing of the Java language is often useful, it sometimes forces one to pay for (or bypass) some complexity. Additionally, Java has only limited support for overloading, which concerns the way rule scheduling is implemented. More specifically, Java does not support multiple dispatch (cfr. Section 2.4.4.2).

6.3.5.1 Java Overloading: Implicitly Check *Static Types* of Arguments

Java supports that one class implements multiple interfaces. This introduces a potential ambiguity in the resolution of calls to overloaded methods.

More specifically, when an argument's class implements two (or more) interfaces, and a method implementation is available for both interface types, one needs an additional resolution rule to decide which method implementation should be triggered. This issue also applies to arguments whose class subclasses a class *c* and implements an interface *i*, when there are overloaded methods available both for parameter types *c* and *i*.

In Java, this issue was resolved by stating that the resolution of calls to overloaded methods is based on the static types of the arguments. Since an object can only be statically typed with one class or interface, all ambiguities can be resolved.

Unfortunately, the resolution rule is overkill in many practical scenario's, including the declarative rule scheduling scenario described in this section. More specifically, the Story Diagram from Figure 6.22 and the Story Pattern from Figure 6.23 map to Java code in which the nodes from the input activity diagram are statically typed as *ActivityNode* instances.

Due to the strong typing rules of Java, one needs to provide a method *transform(node: NamedElement, out: CspContainer)*. One may expect that the overloaded transform methods discussed in the previous section will still be triggered by the *transform(n, ...)* call in the Story Diagram. However, this is not the case: the first parameter of the *transform(node: NamedElement, out: CspContainer)* method matches with the static type of *n*. The dynamic types of the nodes in the activity diagram (*InitialNode*, *ForkNode*, ...) are ignored by the overloading resolution rules.

6.3.5.2 Solution 1: Explicitly Check Types and Cast Arguments

The most straightforward, but least reusable and most unreadable, approach consists of the explicit implementation of a *type-check* conditional in the auxiliary *transform(node: NamedElement, out: CspContainer)* method:

```
if (n instanceof InitialNode) { transform((InitialNode)n); } else
if (n instanceof ActionNode) { transform((ActionNode)n); } else
if (n instanceof DecisionNode) { transform((DecisionNode)n); } else
if (n instanceof MergeNode) { transform((MergeNode)n); } else
if (n instanceof ForkNode) { transform((ForkNode)n); } else
if (n instanceof JoinNode) { transform((JoinNode)n); } else
if (n instanceof FinalNode) { transform((FinalNode)n); } else {
// ignore node: do nothing
}
```

Obviously, the proposed implementation is completely UML metamodel specific. This requires one to test (or in general “maintain”) it for this specific transformation, which is of course undesirable. Interestingly, this approach can be generalized into an extension of Story Diagrams. More specifically, one may define an implicit scheduling operator from which the auxiliary method and its conditional is generated by means of a higher order transformation.

Since that kind of transformations is already discussed in the context of the `«alias»` construct (and in Chapter 7), the following subsection presents an alternative approach.

6.3.5.3 Solution 2: Reflective Interpreter

The UML2CSP transformation relies on a reflective helper method to “supplement” the static overloading rules with dynamic overloading behavior. More specifically, the auxiliary *transform(node: NamedElement, out: CspContainer)* method delegates to a method “callOverloadedMethodDynamicallyForFirstArgument” that can be considered as a kind of interpreter for dynamic overloading behavior.

Essentially, the helper method computes the dynamic types of the arguments as explicit sets of classes and interfaces. The Cartesian product of these sets represents all parameter types of methods that are compatible with the dynamic types of the two arguments. The helper method enumerates this Cartesian product for “guessing” the signature of transformation rules for the two arguments.

The guess is validated by relying on the Java API for calling a method reflectively. If the guess was right, a transformation rule has been executed successfully. Otherwise, an exception is caught and another guess is generated. If no matching *transform* method can be found, the interpreter transfers control back to the *transform(node: NamedElement, out: CspContainer)* method, which terminates silently. As indicated by the Story Diagram from Figure 6.22, other activity diagram nodes may then be matched.

Since the call to the helper operation can also be generated, this approach is as platform independent as the approach relying on the explicit type checks. However, due to the “trial and error” nature, the performance is worse. More specifically, the time complexity of the interpreter is $O(m \star n)$, where m is the number of classes and interfaces defining the type of the first argument, whereas n is that number for the second argument.

For the sample activity diagram described in the problem description from Bisztray [26], the helper is called 18 times and m is 33 on average. Although it seems that the dynamic type of the second argument can be fixed to the *CspContainer* class, it may consist of another (more platform-specific) class at the execution level.

For the architecture described in Section 6.3.1 for example, the dynamic type of the instance consists of 12 classes/interfaces. Since the Java reflective API requires an exact signature match, we rely on the trial and error mechanism for the second parameter as well.

The time complexity of the explicit type-checking approach discussed in Section 6.3.5.2 only depends on the number of activity node types. That number is 7 in this case study and constant in general.

6.3.5.4 Solution 3: Move the “transform” method

As another potential solution, one may want to refactor the transformation model into a form that only requires single-dispatch. More specifically, one may want to move each “transform” method to the class corresponding to its first parameter. For example, it seems to make sense to move the *transform(MergeNode, CspContainer)* from the *UML2CSP* class to the *MergeNode* class. All references to the first parameter would obviously be substituted by references to the *this* object. Since polymorphism would then only be needed on the receivers of the *transform* messages, the limitations of the underlying single-dispatch platform would not lead to erroneous behavior.

Unfortunately, the above design is in conflict with some integration requirements at the implementation level. More specifically, when the transformation needs to be executed in-memory, on the actual repository of a UML 2 tool, the implementations of all repository classes are fixed (since they are part of the industrial CASE tool). Even when custom implementation subclasses can be plugged in the CASE tool (which is the case with MDR [169]), there is no means to add an abstract *transform(CspContainer)* method to the *NamedElement* interface that is referenced by the caller to the transform methods. Still, such a method is needed for type safety in the client code. Due to these complications, this third solution has not been elaborated further. Solution 2 has been used instead.

6.4 Lessons Learned

This section presents a brief overview of the lessons learned from the development of the MoTMoT *UML2CSP* transformation class.

First of all, the case study illustrates that MoTMoT is effectively usable for the transformation of UML 2 models produced by an industrial modeling tool. Although previous case studies already applied MoTMoT to models conforming to different metamodels (UML 1 [267] and Traceability [257]), a UML 2 case study is an especially convincing means to illustrate that MoTMoT is applicable on any MOF metamodel: the complexity of the UML 2 metamodel did not reveal any scalability problems. More specifically, tests indicate that the performance of the generated MoTMoT code is much better than that of the XMI reader from the CASE tool provider. Apart from this evidence of scalability, the case study solution deals with the MOF 2 package merges that are frequently applied in the standard UML 2 metamodel. Even with a proprietary realization of these package merges at the execution level of the input CASE tool's XMI reader, it appears feasible to model the transformation itself in a standard compliant manner.

Secondly, the case study reinforces our confidence in the quality of the UML profile for Story Diagrams. On the one hand, the expressiveness of controlled graph transformation is clearly illustrated and the elaborate descriptions of the Story Diagrams and Story Patterns may be a useful tutorial supporting the adoption of the formalism in industry. On the other hand, the presented transformation model involves several design choices that heavily influence the readability, compactness, and reusability of the diagrams shown in this chapter. For example, several *«loop»* constructs can be refactored into combinations of *«success»* and *«failure»* links. Similarly, some complex Story Patterns can be decomposed into two or more sequentially executed Story Patterns.

According to the modeling style of previous publications on Story Driven Modeling [62], the complexity of the Story Pattern from Figure 6.18 would be tackled by distributing its responsibilities across two states: a first state would take care of the actual mapping from input to output element while a second state would establish all connections that relate to the control flow of the transformation.

In contrast, the presented solution deliberately merges such sequences into one state. The complexity of the Story Patterns is managed by means of “views”. This approach has several advantages: first of all, introducing a sequential execution order where it is not required can be confusing. In terms of the example from Figure 6.18, there is no reason why one would first take care of the model mapping and only then create control flow-related elements. Secondly, when applying transformation views, one can investigate more easily the complete set of edges

that relate to one rewrite node in a coarse-grained step within the transformation process.

The *UML2CSP* transformation model is the first example in which the *reuse* of Story Patterns is illustrated. Approaches relying on calls to complete Story Diagrams force transformation writers to define some artificial methods that have only one state. Instead, the *IncomingTransition2OutputAssignmentExpression* Story Pattern (cfr. Figure 6.13) is used directly from the Story Diagram for transforming decision nodes (cfr. Figure 6.8) and from that for transforming fork nodes (cfr. Figure 6.15). Since no method call is involved, the renaming of variable names needs to be tackled explicitly. To this aim, this chapter used the `<<alias>>` construct. This construct is not restricted to the renaming of node variables: it supports a type-cast operation too.

The case study also confirms some lessons learned from previous chapters. For example, this chapter illustrated once more that a proper use of colors and specific layout patterns can significantly improve the readability of transformation models. While these colors may not give much clues to novice “users” of the language, they decrease the time to understand a Story Pattern already after a very short learning period. Similarly, the layout patterns applied throughout this chapter have no relation to the transformation language semantics. Therefore, these patterns can be optimized completely in terms of the input and output modeling languages. For example, in most Story Patterns presented in this chapter, the input transitions are displayed above activity nodes while output transitions are displayed below.

Interestingly, the case study also exposes the current limitations for modeling transformations in a platform-independent manner. As discussed in Section 6.3.2, the *UML2CSP* transformation class presented in this chapter would have a method *transform(inputElement, outputContainer)* with different parameter types when one would model the transformation completely independent of the target modeling platform. On the one hand, the effect of this concrete problem is rather limited, as the signature can be changed very locally and the effect of its change is managed properly by the type-checks of the underlying Java compiler. On the other hand, one could overcome this limitation by defining a tagged value on metaclasses that have a non-standard realization. In that case, the behavioral models (i.e., the signatures of transformation methods, the Story Diagrams and the Story Patterns) of the transformation would be completely platform independent. The platform specific concern would be localized in the structural part of the transformation model, that is: in a tool-specific specialization of the input metamodel.

6.5 Summary and Outlook

The UML-to-CSP case study reveals the following strengths and weaknesses in the MoTMoT approach.

First of all, a key benefit of MoTMoT is the visual nature of its modeling language. The ability to exploit 2D layout characteristics to improve the mental mapping of a transformation model is a key advantage of visual graph transformation languages such as Story Diagrams.

A second strength of MoTMoT is its conformance with OMG’s MDA standards: not only can the *UML2CSP* transformation model itself be edited with any UML 1 compliant modeling tool, the code generated from that model also consumes standard UML 2 models and produces CSP models that conform to a MOF based metamodel. Moreover, Section 6.3.4 illustrated how models that do not conform to the standard metamodels of their modeling languages can be normalized in-place to conforming versions.

The third and final advantage of MoTMoT that is presented in this chapter is its extensibility. This strength is illustrated by the fact that the new `«alias»` construct (that was introduced while completing our submission to the *UML-to-CSP* case study) is realized again as a normalizing (higher-order) transformation from the extended profile for Story Diagrams towards the original profile.

Unfortunately, but also obviously, the MoTMoT approach has some limitations. First of all, a drawback of the UML profile for Story Diagrams is that a generic UML editor does not provide the same usability features as the dedicated Fujaba editor. This acknowledges a conclusion from Chapter 5: it may be worthwhile to realize a mapping between MoTMoT's UML profile and Fujaba's representation of Story Diagrams. Moreover, it may be industrially relevant to specialize generic UML editors with a plugin that offers specialized behavior such as the auto-completion of variable names in Story Patterns. As a second limitation, some platform specific details can not yet be separated from conceptual transformation models. However, there are various approaches for solving this in future work.

Chapter 7

Copying Subgraphs within Model Repositories

This chapter is based on a paper that has been published in the proceedings of *GT-VMT'06* [267].

The set of operations in graph transformation languages such as Story Diagrams allow one to conditionally create and remove nodes and edges from input graphs. Node attributes can be initialized or updated with information from other attributes, parameters or constants. These operations appear to be too restricted for expressing out-place, endogenous transformations in a concise manner. More specifically, graph transformation lacks an operation for copying subgraphs (multiple connected nodes, including their attributes) to a new location in the host graph. This chapter presents a sub-model synthesis case study that illustrates the need, a syntax and an informal semantics for such an operation. It also discusses how the operation was integrated in the UML profile for Story Diagrams. Finally, it indicates how a “compiler” relying on higher order transformations reuses evaluation code for existing language constructs.

Introduction

One of the main purposes of this thesis is to enable the modeling of transformations at a high level of abstraction while the low-level APIs of mainstream model repositories are interfaced by means of a code generator such as MoTMoT. The previous chapters relied on plain Story Diagrams to model transformation behavior. As Chapters 4 and 5 illustrated, Story Diagrams allow one to model in-place transformations in a concise manner. Moreover, Chapter 6 illustrated that out-place transformations could be modeled elegantly when input and output elements have a different type. This chapter illustrates that out-place transformations between (sub)models that closely resemble one another can be modeled at an even higher level of abstraction.

When using plain Story Diagrams to model how input elements are copied elements, one is modeling at a too low level of abstraction. More specifically, plain Story Diagrams require the *explicit* enumeration of all metaclass properties in attribute assignments on created output nodes. As such, the intent that a complete subgraph needs to be copied, gets lost in the details. At this level, one is using Story Diagrams as a programming language instead of as a modeling language.

Figure 7.1: Conceptual Model of a Meeting Scheduler application.

Instead, one wants to model concisely that *complete* subgraphs should be copied from the input model to the output model. Although one does need to specify which nodes and edges belong to the copied subgraph, the transfer of all node attributes should be implicit. Therefore, this chapter introduces an operator for modeling such subgraph copy operations explicitly. The operator is applied on an example of endogenous, out-place, sub-model synthesis and is formalized by “desugaring” it to standard Story Diagrams.

This chapter is structured as follows: Section 7.1 presents two models of a meeting scheduler system. These models are expressed in two different UML profiles and a part of one model should be *generated* from the other one. When using graph transformation to formalize the model transformation that defines this generation process in Section 7.2, the need for a copy operator becomes obvious. Section 7.3 presents the syntax and semantics of the proposed copy operator as an extension to Story Diagrams. Additionally, the section briefly compares two approaches for extending an existing a Story Diagram engine such as MoTMoT or Fujaba. The next section refers the reader to related work while the chapter concludes with a summary of the contributions and lessons learned.

7.1 Motivating Example Models

Figure 7.1 shows a conceptual model (CM) of a Meeting Scheduler application, specified in UML syntax. At the conceptual level, analysts are free to use constructs such as association classes, views, and other language features. Such features may not be supported directly by the implementation language but they allow one to represent the problem domain as one perceives it in reality as good as possible.

A complete conceptual model contains all relevant nouns and verbs from a problem domain as classes and operations. In order to localize changes to the problem domain, many architectures hide the conceptual model by means of layers. To design such architectures, Rosenberg and Scott [220] propose to model user interface screens as *interfaces* and user interface flow as *services*. Only services are allowed to access *entities*, which are based on the classes in a conceptual model. Figure 7.2 shows a robustness model (RM [220]) of the application under study. Note that the entity *Schedule* corresponds to the class *Schedule* from Figure 7.1.

Figure 7.2: Robustness Model of a Meeting Scheduler application. Such a model contains more technical concepts than a conceptual model. For example, this diagram shows that the login “interface” can only access data about person “entities” through the account “service”.

Figure 7.3: Relation between the UML editor and the underlying model graph.

Figure 7.3 clarifies how the elements from the conceptual modeling diagram shown in Figure 7.1 relate to a typed and attributed graph in the underlying model repository. The tree on the left represents the “containment hierarchy view” from the Meeting Scheduler sample in the MagicDraw UML tool. Node *n1* is of type *Model* and represents the UML model that contains both the application examples and the definitions of the profiles used within these examples. All examples reside in node *n2* of type *UmlPackage* and with name “Examples”.

Node *n3* represents the actual Meeting Scheduler sample. This *UmlPackage* contains node *n4* which represents the conceptual model of the Meeting Scheduler. All its contained classes (like *Attendee*, *Flexibility*, ...) map directly to concepts in the problem domain. The containment relationship between *n1*, *n2*, *n3* and *n4* is realized by means of links *l1*, *l2* and *l3* with label “ownedElement”. These links can be traversed in the other direction (from contained element to container) as well by means of the “namespace” label. Therefore, the underlying graph is not a directed graph. Moreover, it contains cycles: node *n4* (*CM*, the conceptual model of the Meeting Scheduler) is decorated with the “Conceptual Model” stereotype by means of link *l4*. This link can be edited by means of the context-sensitive menu shown in the bottom right corner of Figure 7.3. The “Conceptual Model” stereotype is defined by node *n7* which is contained in node *n6*, representing the package defining the robustness modeling profile. Due to space limitations, *n7* is not shown in Figure 7.3. However, the figure does show a node defining another stereotype: node *n5* represents the definition of the “Foreign Key” stereotype from the profile for physical data modeling.

In the following, it will be shown how the entities in the robustness model can be created automatically from the classes in the conceptual model by means of the subgraph copy operator. The idea is to integrate the approach into model editors such that software engineers can focus on design decisions in the model refinement process rather than performing low-level copy operations manually.

7.2 The CM2RM Transformation

This section discusses the nature of the “Conceptual Model to Robustness Model” (*CM2RM*) transformation by presenting a structural and a behavioral model. The structural model will illustrate how the transformation is related to data from the input and output repositories. The subsection discussing the behavioral model will focus on the application of the subgraph copy operator.

7.2.1 Structural model of the transformation

As stated in the introduction, the structure of a modern model repository is defined by an object-oriented model. More specifically, such a “*metamodel*” represents the language of the models that can be stored in the repository. Since such metamodels define the input and output types of model transformations, they are discussed for the modeling languages used in the running example. Both the conceptual and the robustness models are expressed in the UML. Since the UML profiles that decorate the standard diagrams with a domain specific syntax are defined as UML models as well, the transformation under discussion only needs to interact with UML repositories.

Figure 7.4: Metamodel that allows one to store the transformation with a direct reference to the UML model it transforms. The class “Model” is imported from the UML metamodel.

Figure 7.4 shows a structural model of the *CM2RM* transformation. The interesting fact about this diagram is that one does not have to reason about the distinction between transformations, models, metamodels and meta-metamodels (as defined in MOF [193]) to understand its meaning. It is a traditional class diagram that happens to be used in the context of model transformations but that does not presume any knowledge about the platform-specific repository code that is generated from it.

The *CM2RM* transformation contains a reference to one *Model* (defined in package *org.omg.uml.modelmanagement*) while such a *Model* can be transformed by many *CM2RM* transformations. The *Model* class, its association to the contained UML *Model Elements* (*UmlClass*, *UmlPackage*, *State*, ...) and other concepts from the UML are defined in the UML specification [195]. Since the repositories from popular UML tools are derived from (often even *generated from*) this specification, the class implementing the *Model* concept in Magic-Draw does not define a collection of *CM2RM*s. Therefore, the *UMLmodelOfTransformation* association can only be traversed from *CM2RM* to *Model*. In order to apply the *CM2RM* transformation to the example from Section 7.1, the *applicationModel* reference needs to be initialized with the “Data” model (node *n1* from Figure 7.3).

The *CM2RM* transformation can be parametrized with its *applicationName* attribute. This attribute determines what package inside the UML model will be looked up in order to transform the classes in its contained conceptual model to entities in its contained robustness model. When the *CM2RM* transformation would be applied to the example from Section 7.1 then *applicationModel* would be set to node *n1* from Figure 7.3 while *applicationName* would be set to “Meeting Scheduler”. This would configure *CM2RM* for execution on *n3*.

While the *CM2RM* transformation could contain more methods for more complete case studies, this chapter requires only one which is called “*cmClasses2rmEntities*”. The method does not take any arguments and returns *true* or *false* based on the success of the transformation. The complete behavior of *cmClasses2rmEntities* has already been discussed before [260] but this chapter provides a more comprehensive discussion of the *copy operation* used there.

7.2.2 Behavioral model of the transformation

The behavior of the *cmClasses2rmEntities* method can be modeled in two phases. Firstly, the transformation needs to look up some meta-information for robustness modeling in the UML. Secondly, the classes are copied from the conceptual model to the robustness model and they are marked as entities by decorating them with the proper meta-information. Each of these steps can be implemented as a Story Pattern while the order between the primitives needs to be enforced by a Story Diagram.

Figure 7.5: Primitive graph transformation rule applying the new copy operator.

Figure 7.5 shows the primitive graph transformation rule for phase two. The rule is written in the UML profile for Story Diagrams, into which the proposed copy operator is integrated. The following subsections discuss the semantics of the new operator by means of this example.

7.2.2.1 Finding a Match

The nodes and edges that do not have a «create» stereotype in a primitive story specify a pattern that needs to be found in the input model. The pattern on Figure 7.5 starts from a node representing *CM2RM*'s *applicationModel* property. As stated, this property represents a handle to the input and output UML model of the discussed transformation (like node *n1* from Figure 7.4). Just like the *stereotypeOnCM*, *stereotypeOnRM* and *entityStereotype* nodes, the *applicationModel* node is already *bound*: in fact, attributes of transformation classes are bound during the construction of the transformation object while the stereotype nodes are bound by the first primitive graph transformation rule of *cmClasses2rmEntities*.

From the *applicationModel* node, the rule searches for each recursively contained package with its name equal to the *applicationName* property of *CM2RM*. Such a *UmlPackage* is called *wodnApplication* and it represents the application containing the model that needs to be copied. Variable *wodnApplication* would be bound to node *n3* from Figure 7.4. Note that all nodes and edges are typed and map directly to the class diagrams defining the UML meta-model. The *UmlPackage* nodes that can be reached from the *applicationModel* by recursively traversing outgoing links with association end name *ownedElement* are only bound to the *wodnApplication* node if they in turn contain a specific *cm* node in their outgoing *ownedElement* links. A node is bound to *cm* if it is of type *Model* and contains the already bound *stereotypeOnCM* node in its outgoing *stereotype* links. By specifying that *cm* contains zero or more nodes of type *UmlClass* with zero or more nodes of type *Attribute*, one does not constrain the search for *cm* any further.

7.2.2.2 Copying the Subgraph

The *cm* node needs to be copied since it is decorated with the «copy» stereotype. Apart from the *cm* node, all nodes and links on its outgoing composition path need to be copied as well. Note that all matches on this path are handled since the controlled graph transformation rule that executes this primitive rule marks it as «loop». Without this directive, the primitive rule shown on Figure 7.5 would copy only one matched class and attribute.

Implicitly, all attributes from a copied node are copied along. For example, since it is of type *Model*, the *name*, *isSpecification*, *isRoot*, *isLeaf* and *isAbstract* attributes of node *classInCM* are copied implicitly. For the definition of the *Model* class, its attributes and superclasses, please refer to the metamodel in the UML 1.5 specification [195].

7.2.2.3 Using the Copy

When copying a subgraph, one should always store a reference to the copy. Otherwise, it wouldn't become a subgraph of the host graph but just a standalone graph which may be inaccessible in subsequent graph transformations. The undesirable result would be an output model that would not contain the copy.

Creating a link is a standard graph transformation operation. In the UML profile for Story Diagrams, one needs to specify a link between the nodes that need to be connected and label it with the «create» stereotype. Obviously the name of the link and the name and cardinality of the association ends need to conform to an association between the types of the node. Otherwise, the resulting graph would not conform to the output metamodel. In order to create a link from the *wodnApplication* node to the copy of the *cm* node, one needs an explicit notion of node copies in the graph transformation language.

Instead of representing the copy as a node in the transformation rule, the UML profile for Story Diagrams is extended with an «onCopy» stereotype. By specifying it on the *ownedElement* association end of the «create» link that connects *wodnApplication* with *cm*, one expresses that the link should be created to the copy of *cm* instead of to *cm* itself. When the «onCopy» stereotype would not be specified on *ownedElement* end, one would erroneously specify that the conceptual model needs to be added to the package it already resides in. The robustness model would be missing from the output model.

The «onCopy» construct is also defined in the context of attribute assignments. This allows one to specify that the name of the robustness model, that is a copy of the *cm* node, needs to be changed to “RM”: the attribute assignment on the *cm* node is decorated with the «onCopy» stereotype. Without this stereotype one would change the *name* attribute of the conceptual model.

The «onCopy» construct for «create» links is also applied to decorate all classes in the robustness model with the «entity» stereotype: the association end at the *classInCM* side of the stereotypes link is decorated with the «onCopy» stereotype while the association end at the *entityStereotype* side is left undecorated. The class in the robustness model is indeed a part of the copied subgraph while *entityStereotype* is a node in the original host graph.

The outgoing *type* link of node *a* (of type UML *Attribute*) needs to be copied to the target subgraph as well. A detailed discussion thereof is outside the scope of this chapter. In summary, the rule on Figure 7.5 needs to be extended and an additional loop story is needed. By using multi-objects in combination with the «onCopy» construct one can first create and then query the required traceability data.

7.3 Subgraph Copy operator

This section presents a syntax and an informal semantics for the proposed copy operator as an extension to the UML profile for Story Diagrams. It also compares two implementation approaches to motivate the direction of the ongoing effort.

7.3.1 What

The proposed copy operator consists of the following syntactical constructs:

copy The «copy» construct allows one to specify what node represents the entry point to the subgraph that needs to be copied.

composition Starting from the «copy» node one can specify that a particular match path has composition semantics. Each node and link on this path will be copied.

onCopy The «onCopy» construct can be used to indicate that a particular instruction needs to be executed on the copy of an element instead of on the element itself. The construct is defined on (1) association ends of «create» links and (2) attribute assignments.

1. By specifying «onCopy» on the source (or target) end of a «create» link, one specifies that the link needs to be created from (or to) the copy of the node at that association end. This construct enables one to create target subgraphs with cyclic links whereas the *composition* construct discussed above only supports a hierarchical structure.
2. An assignment on an attribute from a node on the composition path, that is marked as «onCopy», is executed on the attribute from the copy of this node instead of on that from the node itself.

Not every application of these directives results in a valid use of the copy operator. Therefore, the following new well-formedness rules (WFRs) are defined for the UML profile for Story Diagrams:

- At least one link should be created from the host graph to a node from the copied subgraph. More specifically, at least one link should be created to the «copy» node or a node on its outgoing composition path.
- The «onCopy» instruction should only be applied (1) on attributes inside a copied node, or (2) on association ends connected to a copied node.
- A node should be part of at most one composition. Otherwise, it would be ambiguous what should be the container of such node's copy.

The following OCL specification formalizes the first WFR in OCL:

```

1 context Class
2   -- Return transitive closure of the "ownedElement" links starting from s
3   def: let ownedElementTC(s: Set (ModelElement)): Set (ModelElement) =
4     if s->includesAll(
5       s->select (me1 |
6         me1.oclIsKindOf (Namespace)
7       )->collect (me2 |

```

```

8      me2.oclAsType (Namespace)
9      ).ownedElement->asSet ()
10  ) then s
11  else ownedElementTC (
12      s->union (
13          s->select (me1 |
14              me1.oclIsKindOf (Namespace)
15          )->collect (me2 |
16              me2.oclAsType (Namespace)
17          ).ownedElement->asSet ()
18      )
19  )
20  endif
21
22  -- Return from a primitive story all nodes that will be copied
23  def: let allCopiedNodes(s: Set(Classifier)): Set(Classifier)=
24      s->select(c| -- Return all classes
25          hasStereotype(c, "copy") or -- that have a <<copy>> stereotype
26          c.association->exists(end| -- or connected to
27              end.association.connection->exists(end2| -- an association
28                  end2<>end and -- of which the other end
29                  end2.aggregation=AggregationKind::composite -- is of type composite.
30              )
31          )
32  )
33  -- Actual WFR: as soon as the <<copy>> instruction is issued, the copied sub-
34  -- graph needs to be connected to the host graph by means of a <<create>> link
35  inv:
36  let trfoPkgNodes: Set(Classifier) =
37      ownedElementTC(Set{self.namespace})->select (element |
38          element.oclIsKindOf(Classifier)
39      )->collect(class |
40          class.oclAsType(Classifier)
41      )->asSet in
42  let copiedNodes: Set(Classifier) = allCopiedNodes(trfoPkgNodes) in
43  let nonCopiedNodes: Set(Classifier) = -- trfoPkgMEs minus copiedNodes
44      trfoPkgNodes->reject(el|
45          copiedNodes->exists(copiedNode|
46              el=copiedNode -- Reject elements that are copied (set 'minus').
47          )
48      ) in
49  hasStereotype(self, "copy") implies -- When applying the copy instruction,
50  nonCopiedNodes.association->exists(end| -- the non-copied nodes should be
51      hasStereotype(
52          end.association, -- connected to an association
53          "create") and -- representing a <<create>> link
54          end.association.connection->select(end2| -- and containing
55              end<>end2 -- another end that
56          ).participant->exists(copiedNode| -- is connected to a node
57              copiedNodes->includes(copiedNode) -- that *is* copied.
58          )
59  )

```

The specification is defined within the context of the *Class* class from the *Foundation::Core* package of the UML metamodel. Every instance of that metaclass needs to respect the invariant defined from line 56 onwards. One can use the OCLE tool [42] to confirm that the “cm: Model” node from the transformation rule in Figure 7.5 respects this invariant. The constraint makes use of three OCL helper attributes defined on line 43 to 49 and 50. The *trfoPkgNodes* attribute represents all nodes from the copy transformation rule under study. The *copiedNodes* and *nonCopiedNodes* attributes divide this set of nodes into the nodes that will or will not be copied respectively. These attributes are defined using the helper operations specified on line 10 and 30. The OCL specification of the latter two WFRs is left out due to space considerations but can be obtained from the author.

Figure 7.6: From left to right: two valid rewrite rules and an invalid one.

Figure 7.6 clarifies the third WFR. In the rightmost rule, it is unclear whether the c 's contained by the b 's from $ab1$ should be copied, or those contained by the b 's from $ab2$, or both. The leftmost rewrite rule illustrates how one can unambiguously specify that for the b 's from $ab2$ the contained c 's should be copied while this is not the case for those from $ab1$. The middle rewrite rule illustrates that within one rewrite rule one can use multiple «copy» nodes as long as their composition paths do not overlap.

7.3.2 How

Two implementation approaches have been investigated: a direct model-to-code transformation approach and a model-to-model transformation approach. All related artifacts are publicly available in the MoTMoT project [178].

The direct model-to-code approach involves the extension of the dynamic content templates that have been discussed in Section 5.2.2.3 from Chapter 5. The “basic” templates only handle the code generation for the Zündorf Story Diagram constructs that have been discussed in Section 1.3.4 from the introductory chapter. At a very high level of abstraction, the generated code should implement the following additional tasks:

1. collect all nodes matching the composition path specified in a copy rule,
2. in the case of a complete match: (a) copy these nodes, including all their attributes, and (b) execute «onCopy» attribute assignments,
3. maintain a map of traceability links between nodes and their copies,
4. use the traceability map to create the composition links between the copies as soon as all of the copy nodes have been created,
5. create «onCopy» links using the same approach.

In practice, the complexity of the templates reached an unacceptable level after implementing step (iv). Even if we would have been able to implement the complete algorithm in the templates, their complexity would have compromised the maintainability of these artifacts.

Therefore, we shifted to a model-to-model transformation approach that leaves the code templates unchanged. Chapter 8 discusses in more detail how Story Diagrams can be used to transform models conforming to the profile discussed in Section 7.3.1 into models conforming to the profile without the copy operator. The complete transformation is still complex but thanks to the use of an intermediate layer and the modularity mechanisms of Story Diagrams, the complexity can be decomposed into manageable parts. Apart from the facilities for managing the transformation complexity, the model-to-model transformation approach has the following benefits:

- It does not involve a further investment into code specific to the MDR/JMI platform. Migrating the templates to platforms such as EMF does not become harder than before.
- With reasonable effort, it should be possible to deploy the story diagrams that are generated by the model transformation on other Story Diagram platforms such as Fujaba.

7.4 Related Work

Subgraph copying was first investigated in the context of *hierarchical* graph transformation. This work assumes that one can decompose the transformed graphs into “frames” where edges are not allowed to cross frame boundaries. Drewes, Hoffmann and Plump acknowledge that nested visual languages like the UML require a more flexible decomposition mechanism. However, they rely on their assumption to prove that particular rewrite rules respect a set of grammatical constraints [66].

Although the hierarchical approach presents the interesting idea of automatically copying all edges between the nodes in a frame, it should be extended for performing copy operations in a more general sense. An «onCopy» construct such as the one presented in this chapter could be defined to specify that, for example, the copy of a subgraph should not contain particular edges while including others that do not originate from the source subgraph. Another limitation of the hierarchical approach is that frames are not proposed to be defined on a rule by rule basis. Hoffmann et al. tackled this issue by allowing “shape grammars” to define the structure of a frame variable in the scope of a rewrite rule instead of in the scope of the complete rewriting system [20].

Figure 7.7: Project Structure before running *CM2RM*.

7.5 CM2RM Revisited

This section presents an extension of the simplistic *CM2RM* transformation discussed above and in [267] to illustrate where the copy operator may be improved. To this purpose, subsection 7.5.1 refines the desired transformation behavior. Subsection 7.5.2 clarifies how the copy operator already supports these additional requirements. Nevertheless, as subsection 7.5.3 illustrates, an extension of the operator would enable one to model the transformation at an even higher level of abstraction. To make the rather abstract concepts tangible, the transformation requirements are formulated in terms of the running example. However, it should be clear that neither these requirements, nor the nature of the copy operator is specific to the meeting scheduler case study.

7.5.1 Extended Requirements for *CM2RM*

This subsection revisits the example application models presented in Section 7.1. By discussing additional mapping rules between the example conceptual and robustness models, we derive additional requirements for the generic *CM2RM* transformation. Figure 7.7 shows a screenshot of a modeling environment in which three example application models are loaded.

As explained before, all application models are contained in the package “Thesis Examples”. Before executing the *CM2RM* transformation, the “Meeting Scheduler” example only contains a conceptual model. The “Human Resources” example contains a conceptual model,

Figure 7.8: Example project after running *CM2RM*.

a physical data model (see Section 1.2.1.1), a model for object/relational mapping and a robustness model. This can be seen on the “project tree” shown on the left side of the screenshot and on the upper diagram on the right side of the screenshot.

The goal of the *CM2RM* transformation was to generate a robustness model from a given conceptual model. As discussed in Section 7.2, the transformation can therefore be parametrized with an application package. In the case a *CM2RM* transformation would be parametrized with the package containing the meeting scheduler model, it should be able to generate a robustness model from the contained conceptual model shown on the bottom right of Figure 7.7. This robustness model, and all its elements, should be added to the “Meeting Scheduler” package.

Figure 7.8 displays a screenshot after thus applying the *CM2RM* application. The project tree view, shown on the left of the figure, now contains a robustness model called *RM*. The right side of the figure shows some of its contained entities that are generated from the classes in *CM*. It should be noted that the associations from the conceptual model are copied to identical associations in the robustness model. Similarly, inheritance links are copied but these are not shown on Figure 7.8 due to presentation considerations. An extended version of the *CM2RM* transformation may handle the flattening of association classes as well. The interested reader is referred to the work of Gogolla [102, 37].

Two entity nodes, called “Meeting Scheduler” and “Schedule” are shown as expanded

Figure 7.9: Different kinds of attribute types need to mapped to the robustness model.

classes. This is done to illustrate that the attributes and the types are copied from the conceptual model to the robustness model as well. Two types of Attribute nodes have to be copied: attributes in the conceptual model can either have a type that is contained in the conceptual model as well. Such attributes are said to have an *internal* type. Other attributes in the conceptual model may have a “library type”, that is: those attributes have a type that is contained in a package that is not contained within the conceptual model. Such attributes are said to have an *external* type.

The difference between internal and external attribute types relates directly to the transformation from conceptual models to robustness models. For attributes with a type that is internal to the conceptual model, the corresponding attribute within the robustness model needs to have a type that is internal to the robustness model. Thus, in that case, the original attribute and its copy are associated with another type node. In contrast, when an attribute has an external type, its corresponding copy should point to exactly the same library type.

Figure 7.9 visualizes the difference between these two kinds of Attribute types. The *from* and *to* attributes from “Time Interval” on Figure 7.9 (a) are internally typed: “Moment” is contained within *CM* as well as in *RM* and the attributes use the type within their own model. The *level* attribute from “Flexibility” on Figure 7.9 (b) is externally typed: “Integer” is contained within the library model *LM* and both copies of the *level* attribute share that type.

These specific Attribute type requirements are satisfied by the robustness model from Figure 7.8 too. More specifically, the *name* attribute from “Meeting Scheduler”, and the *acceptableLimit* and *cancelLimit* attributes of “Schedule” are internally typed. These attributes are contained within nodes of type `<<Entity>>`. Still, their type link refers to *string* and *int* nodes that are contained in an external package. On Figure 7.8, the type attribute of “Schedule” is an example of an internal attribute type, since that “Schedule” enumeration is contained in the robustness model as well as in the conceptual model.

This transformation problem challenges the copy operator, as the copying of *externally* typed attributes requires links that cross subgraph boundaries. This kind of copy operations is not supported by the hierarchical graph transformation approaches that are discussed in Section 7.4. In contrast, the solution can be modeled by means of the copy operator, as illustrated by Section 7.5.2.

The final extension to the case study that is shown on Figure 7.8 relates to the node “Schedule Type”. This node is a UML enumeration, defining two enumeration literals with value “personal” and “professional”. To copy nodes of type Enumeration, the transformation model from Section 7.2 needs to be extended.

Figure 7.10: Adding support for copying enumerations, associations and generalizations.

7.5.2 Extended Transformation Model

This section briefly clarifies what extensions to the Story Diagram and Story Patterns presented in Section 7.2.2 are required to support the additional requirements presented in Section 7.5.1.

Figure 7.10 shows an updated version of the Story Pattern shown on Figure 7.5. As one may expect, one needs to extend the scope of the subgraph that needs to be copied. More specifically, one needs to specify that all *Enumeration* nodes contained within the subgraph representing the conceptual model need to be copied to the robustness model subgraph too.

This is realized by adding an additional composition path from the node representing the conceptual model (*cm*), across the nodes representing the contained *Enumeration* elements (*enumInCM*), to the literal values contained in the enumerations. Figure 7.10 also shows that an additional composition path is needed for copying *UML Association* nodes and their ends too. Finally, *UML Generalization* nodes (representing inheritance links), are copied similarly: the composition from *cm* to *cmGeneralization* expresses that such elements need to be copied to the target subgraph as well.

Figure 7.11 shows the Story Pattern that manages the flow between all the rules that are needed to support the requirements concerning the copying the two different kinds of attribute types and concerning the copying of associations. The main difference with the simplistic control flow from Section 7.2 is that four additional rules are triggered: one rule for handling internal attribute types, one for handling external attribute types, one for copying association end participants, and one for setting the parent and child links of the copied *Generalization* nodes. All four rules need to be executed for all possible matches, as indicated by the applications of the `<<loop>>` stereotype.

Finally, Figure 7.12 shows the Story Pattern for handling internal attribute types. As defined in the previous subsection, such attribute types consist of classifiers within the conceptual and robustness model respectively (in contrast to library types that are contained in external packages). The Story Pattern represents the attribute within the conceptual model as the rewrite node with name “*cmAttrWithType*”. This node is connected to its copy, “*copy-OfTypedAttr*”, by means of the traceability link *a2a_internal*. By construction, this copy is

Figure 7.11: Control flow supporting the additional mapping requirements.

Figure 7.12: Story Pattern for handling internal attribute types.

contained within the robustness model. As indicated by the `«create»` link from *copyOfTypedAttr* to *entity*, the type of that attribute within the robustness model is set to a particular entity within the robustness model. By following the link from this *entity* node to its copy *cmClass*, and by investigating the attached *type* link that starts from the *cmAttrWithType* node, one can learn that this *entity* node indeed represents the copy of the type of the attribute within the conceptual model.

7.5.3 Lessons Learned

This section points to the current limitations of the copy operator and how these may be overcome in the future.

7.5.3.1 Traceability Metamodel

Note that the pattern shown on Figure 7.12 relies on a general purpose metamodel for *n*-ary traceability links. With this metamodel, a link can be connected to an arbitrary number of nodes. After realizing the complete case study with this quite expressive metamodel we are considering to model the same transformation with a traceability metamodel for binary traceability links.

The rationale is that we have not needed support for *n*-ary links but the design of the metamodel has a negative impact on the understandability of the Story Patterns. For example, Figure 7.11 shows that the state associated with the Story Pattern shown on Figure 7.12 needs to be associated with a constraint that enforces all nodes connected by a link to be different. In case the nodes would be connected to one node of type *BinaryLink* by means of explicit *source* and *target* links, these constraints would be satisfied automatically.

Another sign over over-design is related to the nodes of type *Node* within Story Patterns such as the one shown on Figure 7.12. These nodes can be useful when elements from different domains need to be distinguished by an attribute “role” of type string. Again, this powerful feature was not needed in this simple case study. Therefore, one may want to use a metamodel in which the *Link* metaclass is associated with the source and target elements directly.

To conclude, this previous subsection has shown that the proposed approach supports subgraph copies involving cyclic links too. However, it seems desirable to improve the way in which traceability links are represented.

7.5.3.2 Copying links between Copied Elements

Three of the four new Story Patterns from the extended flow shown on Figure 7.11 contain a common modeling pattern. More specifically, they rely on traceability patterns between two copied nodes in order to create a link between these nodes.

In the case of Figure 7.12, for example, the *a2a_internal* link is used to navigate from an attribute, “*cmAttrWithType*”, in the conceptual model to its copy in robustness model. Similarly, the *class2entity* link is used to navigate from the type of *cmAttrWithType* to its copy in the robustness model. If such a match is found, a *type* link is created between the two nodes in the copied subgraph. The same pattern is used in the Story Patterns for handling the *participant* link of association ends and for handling the *parent* and *child* links of generalizations.

In general, these Story Patterns *copy a link between copied elements*.

Figure 7.13: Extension of Copy Operator with `<<rematch-node-and-copy>>` construct for links.

This case was already known at the time the copy operator was emerging [254] but has not been included in the first “release” of the operator, as published in [267] and presented in Section 7.3.1. In contrast, we decided to evaluate the “minimal” extensions from Section 7.3.1 first and introduce other constructs only when their need would persist afterwards. The language design rationale is that each new operator increases the learning curve of the language.

In this context, the trade-off between the cognitive dimensions of *consistency* and *diffuseness*, as discussed in taxonomy section 2.4.2, should be considered carefully. More specifically, one can easily introduce a new construct that enables one to enforce the copying of a link between copied elements within Story Patterns such as Figure 7.12. Figure 7.13 shows how such an extension could look like, when applied on that running example.

The Story Pattern contains two new links, when compared to the one shown on Figure 7.5. Both links are decorated with the `<<rematch-node-and-copy>>` stereotype:

- the link between *cmAssEnd* and *classInCM* expresses that from the source subgraph, the *participant* link between an association end and its attached classifier should be copied to the target subgraph. This link makes the behavior from the fifth state in the Story Diagram from Figure 7.11 implicit.
- the link between *a* and *classInCM* expresses that from the source subgraph, the *type* attribute between an attribute and its classifier should be copied to the target subgraph. This link makes the behavior from the third state in the Story Diagram from Figure 7.11 implicit. Consequently, it is semantically equivalent to iteratively executing the Story Diagram for internal attribute copying, as shown on Figure 7.9.

It should be stressed that the semantics of the `<<rematch-node-and-copy>>` construct is different from a `<<create>>` link with two `<<onCopy>>` ends. More specifically, when one would add the latter kind of link to the Story Pattern shown on Figure 7.5, one would create a robustness model within which every attribute would be typed with the entity in which it is contained. In contrast, the `<<rematch-node-and-copy>>` implies that the attached nodes are

rematched before considering to their copies. This motivates the rather complex name of the construct.

Coming back to the “*consistency* versus *diffuseness*” trade-off, the variant of the copy operator presented in subsection 7.5.2 is oriented towards language consistency: the meaning of the Story Diagram shown on Figure 7.12 can be inferred from general purpose language constructs. However, the specification is quite diffuse: a high number of graphic entities is needed to express the traceability relationships between elements in the source and target subgraphs.

Inversely, the extended variant of the copy operator presented in this section is intended to minimize such language diffuseness: Figure 7.13 contains a minimal number of graphic entities to express quite complex copying behavior. However, the `«rematch-node-and-copy»` construct is not consistent with existing Story Diagram constructs. More specifically, since it has no explicit relation to the `«create»`, composition, or `«onCopy»` constructs, its meaning cannot be inferred automatically when learning the extended modeling language.

7.6 Summary and Outlook

This chapter introduced a graph transformation operator for subgraph copying. The operator allows one to define refinements on models conforming to UML profiles in a concise manner. More specifically: copying model elements from one subgraph to another one, changing attribute values of copied elements and attaching links to the copied elements can be done in one rewrite rule. The copied elements can be accessed in subsequent Story Patterns by relying on traceability link navigation. The operator has been integrated in Story Diagrams, a controlled graph transformation language with a wide user base. The extension has been implemented in the UML profile for Story Diagrams such that any UML 1.5 compliant editor can be used to *model* model transformations.

A moderately complex transformation from conceptual models to robustness models has indicated that a general purpose traceability metamodel may lead to undesirably complex rewrite rules. Instead of changing the metamodel to another, more simple one, we intend to parametrize the copy operator such that domain specific metamodels for traceability can be used. That approach would enable one to use a simple metamodel when possible and a more complex one when needed.

An analysis along two cognitive dimensions motivated the minimalistic design of the proposed language extension. This analysis illustrated this chapter did not claim to have found *the* ultimate way to model subgraph copies. Instead, several alternatives are being investigated. However, this investigation is quite focused since all alternatives build upon the language core of Story Diagrams.

This chapter presents a rather specific sub-model *synthesis* case study. However, the copy operator can be used for transforming any typed graph with edge labels and attributed nodes. More specifically, it can be used for implementing *refactorings*.

Van Eetvelde has proposed the use of graph variables and cloning for raising the abstraction level of graph transformation rules in this context [69, 68]. This work extends the case study from Hoffmann [117] by considering the attributes and links from syntax nodes within method bodies in more detail. Applying the copy operator for modeling refactorings such as *Push Down Method*, in terms of a metamodel for Java seems quite promising [46], but the complete transformation has not been modeled yet.

In general, more case studies need to be solved to get a better understanding of the applicability and limitations of the proposed approach to transformation modeling. As part of my ongoing work, I am applying MoTMoT in the context of a graduate-level course on modeling and transformation techniques [124]. Students have already modeled UML refactorings such as *Encapsulate Field*, *Extract Interface* and *Flatten Composite State* [92, 252]. Interestingly, several students that were not introduced to the the work presented in this chapter independently asked how copy operations could be modeled more concisely [133, 166]. Another student group did work in the context of this chapter. More specifically, this group has made a small extension to the *CM2RM* transformation and has shown insight in the supportive engine [15].

Chapter 8

Copy2GT: A Higher-Order Transformation Model

This chapter presents a contribution that has not yet been discussed in a dedicated paper.

The previous chapter motivated the relevance of a *copy* operator and presented how it can be integrated syntactically in the UML profile for Story Diagrams. The chapter described the behavioral semantics of the operator in natural language and at a high level of abstraction. This chapter formalizes the denotational semantics of the operator by means of a higher order transformation. This higher order transformation realizes a *desugaring* (cfr. Section 2.2.1) from transformation models that apply the operator to transformation models that are more machine oriented.

More specifically, the transformation models that are output of the higher order transformation can be mapped to executable code by a copy operator independent version of MoT-MoT. Additionally, output transformation models can be mapped easily to Fujaba, another Story Diagram execution engine that has been developed without taking the copy operator into account.

The result of this chapter is useful since it makes the copy operator executable. This is essential for evaluating its strengths and limitations on more complex transformation problems than the one presented in the previous chapter. Moreover, the higher order transformation itself tackles a complex transformation problem. Since that challenges the scalability of visual transformation modeling in general, it is another valuable evaluation of the proposed techniques.

8.1 Introduction

Formalizing a programming language by means of a denotational semantics consists of mapping the constructs of that language to other constructs whose semantics is assumed to be well-understood. The latter constructs are part of the so-called “semantical domain” [139]. In this chapter, the language construct to be formalized is the copy operator. The semantical domain consists of the Story Diagram formalism without that operator. This variant of Story Diagrams has been introduced in Section 1.3.4 and is formally defined by Zündorf [282].

In the remainder of the text, we will refer to the Story Diagram variant that includes the copy operator as the *Copy* language. The variant without the copy operator will be called the *GT* language. Since both languages are realized as UML profiles, the metamodel of both languages is the same. More specifically, both *Copy* models and *GT* models conform to the UML 1.5 metamodel. Thus, the transformation design is endogenous.

Since *Copy* and *GT* models have a lot in common, this chapter defines the transformation from *Copy* to *GT* by means of an in-place model transformation. Thus, one only needs to model what parts of the *Copy* model need to be changed in order to produce a *GT* model. In the case of an out-place design, the transformation model would have to enumerate explicitly which parts of the *Copy* models are preserved in the *GT* models.

Obviously, this chapter will model the transformation from *Copy* to *GT* models as a set of Story Diagrams. This not only proves we have been “eating our own dog food” [274, 238], it also results in additional lessons learned about the scalability of the basic Story Diagram approach. This chapter is structured as follows: Section 8.2 presents the relation between *Copy* and *GT* models in more detail, Section 8.3 presents some Story Diagrams modeling the transformation between such models and Section 8.4 presents the lessons learned from the construction of this first Higher Order Transformation in the domain of controlled graph transformation.

8.2 Copy2GT: *Desugaring* the profile for Story Diagrams

Before presenting the fine-grained mapping rules between models from the *Copy* language and those from the *GT* language, this section presents an example application of the higher order transformation. Subsection 8.2.1 discusses the application of *Copy2GT* on the first order *CM2RM* transformation from the previous chapter. This example should be considered as a concrete use case scenario for the *Copy2GT* transformation. Subsection 8.2.2 generalizes that “sample run” into mapping rules that are independent of the first order transformation that is used as input for *Copy2GT*.

8.2.1 Example Input/Output Models

Subsection 8.2.1.1 presents the *CM2RM* Story Diagram that is generated by the higher order *Copy2GT* transformation whereas Subsection 8.2.1.2 discusses some of the generated Story Patterns.

8.2.1.1 Story Diagram

Figure 8.1 shows a Story Diagram that models the control flow of the *CM2RM* transformation at a very low level of abstraction. In fact, the rewrite rules that are controlled by this flow only apply the primitive Story Pattern constructs discussed in Section 1.3.4.

Obviously, such Story Diagrams are not to be written manually. Instead, the Story Diagram shown on Figure 8.1 is generated from the Story Diagram shown on Figure 7.11 from the previous chapter. Section 8.3 describes how the supportive transformation from Story Diagrams to Story Diagrams is realized by means of Story Diagrams too.

Although the diagram from Figure 8.1 is significantly more verbose than the corresponding high level variant from Figure 7.11, it is also much more explicit about what is happening

“behind the scenes”. More specifically, the second state (i.e., the state in which the copy operator is applied) has been expanded into 33 automatically generated states. Each of these states corresponds to a Story Pattern that models a particular aspect of the copying process. The four final states of the transformation are preserved since their behavior is already modeled in the *GT* language (i.e., as plain Story Diagrams).

Figure 8.1 highlights the 33 generated states in clusters of light and dark-gray. The layout of the diagram is designed as follows: the top-most cluster of dark-gray nodes contains all states required to copy the model element representing the conceptual model. As discussed in the previous chapter, this element of type *Model* acts as a container for UML classes, enumerations, associations and generalizations. The four clusters of gray nodes that are displayed on the left of Figure 8.1 handle the copying of these contained elements:

- the upper cluster of light-gray nodes contains all states required to copy the contained elements of type *Generalization*,
- the following cluster of dark-gray nodes contains all states required to copy the contained elements of type *Association* and *AssociationEnd*,
- the following cluster of light-gray nodes contains all states required to copy the contained elements of type *Enumeration* and the contained *EnumerationLiteral* elements,
- the lower cluster of dark-gray nodes contains all states required to copy the contained elements of type *UmlClass* and the contained *Attribute* elements.

The order in which these clusters are executed could be altered without changing the behavior of the transformation. In fact, an interesting extension to Story Diagrams would be the use of the *Fork* and *Join* constructs of activity diagrams to express that these transformation states may be executed in parallel.

The diagram layout emphasizes the recursive nature of the transformation behavior. More specifically, each cluster follows a fixed pattern:

select an element from its container in the source subgraph,

check whether the element is already mapped to a copy,

generate a copy if needed,

manipulate the copied element,

recursively apply this pattern on the contained elements.

Figure 8.2 shows a fragment of Figure 8.1 in more detail. More specifically, the states related to the copying of enumerations are shown. The uppermost state iterates over all *Enumeration* elements contained within the conceptual model. The second state checks whether such an element is already copied. The outgoing «failure» transition ensures a copy is generated when needed.

When a copy is present, the outgoing «success» transition ensures that the rewrite rule performing the side effects on the copy is triggered. As indicated by the state name, these manipulations of the copied elements relate to both the *cm* and the *enumInCM* nodes from

Figure 8.2: Control flow of the rules for copying *Enumeration* and *EnumerationLiteral* elements.

Figure 8.3: Selecting all classes that need to be copied.

the Story Pattern shown on Figure 7.10. More specifically, the state *handleOnCopy_cm_EnumsFromCMClonedForCopy_enumInCM* ensures that the copied enumerations are added to the robustness model (which is a copy of the conceptual model). The automatically generated name of this state is based on the name of the edge between the *cm* and *enumInCM* nodes from the Story Pattern shown on Figure 7.10.

The transition to the *select_literal_AsPart* state realizes a recursive step. More specifically, the four states shown on the bottom left of Figure 8.2 realize the “*select, check, generate, manipulate*” pattern described above on all elements contained within an enumeration.

8.2.1.2 Story Patterns

Figure 8.3 shows the Story Pattern that models which classes need to be copied. It expresses that this set consists of those classes that are recursively contained within the conceptual model.

Once more, one should keep in mind that the Story Pattern is *generated* from the “input rewrite rule” shown on Figure 7.10. Thus, a transformation writer has already defined the scope of the desired copy operation by means of the composition construct from the *Copy* language. It would not make sense for the same transformation writer to redefine this scope by means of more low-level Story Patterns such as the one from Figure 8.3.

Instead, the transformation engine will apply the higher order *Copy2GT* transformation to generate such Story Patterns automatically. Still, Story Patterns such as the one from Figure 8.3 enable one to investigate the precise behavior of a model in the *Copy* language. This may be useful for those that want to learn the *Copy* language by relying on knowledge of the

Figure 8.4: Class and Enumeration related view on the input rewrite rule in the Copy language.

Figure 8.5: Generated Story Pattern for Generating nodes of type *UmlAssociation*.

GT language (i.e., plain Story Diagrams). Similarly, analysis and/or execution infrastructure for the GT language can thus be applied for the Copy language too.

Figure 8.4 shows a view on the *input* rewrite rule. This view is more focused than the complete rule specification shown on Figure 7.10. More specifically, it only shows the nodes for copying classes and enumerations. The *ClassesFromCM* link from the Story Pattern shown on Figure 8.3 is generated from the composition link shown on Figure 8.4. Similarly, the *classInCM* node on the rule from Figure 8.3 is generated from the corresponding node shown on Figure 8.4.

Also note that in the “output” Story Pattern shown on Figure 8.3 the *cm* node is bound whereas it is unbound on the “input” Story Pattern shown on Figure 8.4. This is due to the fact that the “output” Story Pattern in the GT language is controlled by the low-level Story Diagram shown on Figure 8.1 whereas the “input” Story Pattern in the Copy language is controlled by the more compact Story Diagram shown on Figure 7.11.

More specifically, at the time that the Story Diagram from Figure 8.3 is evaluated, the *cm* node has already been matched by the second state shown on Figure 8.1. In contrast, within the flow shown on Figure 7.11, the *cm* node shown on Figure 7.10 has not been matched before.

Figure 8.5 shows the Story Pattern for generating a copy of a *UmlAssociation* element while Figure 8.6 shows the corresponding Story Pattern for *Attribute* elements. The Copy2GT transformation generates such Story Patterns for all elements that need to be copied. It should be noted that at the level of the GT language, all node attributes need to be set explicitly.

Figure 8.6: Generated Story Pattern for Generating nodes of type Attribute.

Figure 8.7: Story Pattern for the second state of the *CM2RM* transformation, after applying *Copy2GT*.

It is up to the *Copy2GT* transformation to navigate from the elements upon which the copy operator is applied to the MOF classes defining the types of the copied elements. Using this metamodel information, the *Copy2GT* transformation infers, for example, that in the Story Pattern of Figure 8.5, the copy of the existing *UmlAssociation* node needs to be initialized with the following properties: *isRoot*, *isLeaf*, *isAbstract*, *isSpecification* and *name*. Similarly, as shown on Figure 8.6, the generated Story Pattern for *Attribute* nodes initializes the *copyOf_a_created* node with the following properties of the *a* node: *initialValue*, *name*, *visibility*, *isSpecification*, *multiplicity*, *changeability*, *targetScope*, *ordering* and *ownerScope*.

As another example of the desired input/output behavior of the *Copy2GT* transformation, consider the Story Pattern shown on Figure 8.7. The pattern models the behavior of the second state of the *CM2RM* transformation in the *GT* language. From the perspective of the *Copy2GT* transformation requirements, it should be noted that the corresponding Story Pattern in the *Copy* language contains, among other things, an *«onCopy»* attribute assignment on the *cm* node. This attribute assignment enforces that the name of the robustness model is set to “RM”.

However, in the *GT* domain, the node representing the robustness model does not yet exist in the second state of the transformation flow. Thus, the attribute assignment must be applied in a Story Pattern that is scheduled later in the control flow. The *Copy2GT* transformation

Figure 8.8: Realization of $\llcorner\text{onCopy}\lrcorner$ behavior on $\llcorner\text{create}\lrcorner$ links.

ensures the assignment is executed in the state where the robustness model is generated.

The final input/output example of the *Copy2GT* transformation relates to the $\llcorner\text{onCopy}\lrcorner$ construct for association ends. Consider the $\llcorner\text{create}\lrcorner$ link between *classInCM* and *entityStereotype* on the Story Pattern from Figure 8.4. As discussed in the previous chapter, the $\llcorner\text{onCopy}\lrcorner$ stereotype on the association end attached to *classInCM* ensures the $\llcorner\text{entity}\lrcorner$ stereotype is attached to elements from the target subgraph (i.e., entities in the robustness model) instead of to elements from the source subgraph (i.e., classes in the conceptual model).

Figure 8.8 shows the Story Diagram that models how such manipulations of the copied subgraph are handled in the *GT* domain. The pattern shows that the $\llcorner\text{bound}\lrcorner$ node that represents the *copy* of the *cm* element should be used as the end of the $\llcorner\text{create}\lrcorner$ link.

Although this approach may seem straightforward, it is rather challenging from a transformation modeling perspective. More specifically, generated control flows, such as the one shown on Figure 8.1 need to schedule rules such as the one shown on Figure 8.8 in the proper scope and at the right time. This is a challenging requirement, especially in the case of $\llcorner\text{create}\lrcorner$ links with an $\llcorner\text{onCopy}\lrcorner$ on both ends. In that case, the link between the two copies needs to be created directly after the creation of the second copy.

8.2.2 Mapping Rules

This section summarizes the mapping rules that generalize the example driven requirements discussed in the previous section. Subsection 8.2.2.1 describes the mapping from a rewrite node carrying the $\llcorner\text{copy}\lrcorner$ construct to a set of more low-level Story Patterns. Subsection 8.2.2.2 briefly discusses how the composition path from the copy operator can be mapped to similar, low-level Story Patterns, recursively. Subsection 8.2.2.3 discusses the mapping of the $\llcorner\text{onCopy}\lrcorner$ construct for attribute assignments. Finally, Section 8.2.2.4 discusses the mapping of the $\llcorner\text{onCopy}\lrcorner$ construct for links too.

8.2.2.1 Desugaring $\llcorner\text{copy}\lrcorner$

A rewrite node n_{copy} carrying the $\llcorner\text{copy}\lrcorner$ construct expresses that its matched element needs to be copied. Within a Story Pattern, such a rewrite node should be preserved by the mapping between *Copy* and *GT*. Obviously, its $\llcorner\text{copy}\lrcorner$ stereotype cannot be preserved in the *GT* domain, but all other properties are preserved. Additionally, a “*check and generate if needed*” pattern should be present in the *GT* domain:

- a *check*-state, associated with a Story Pattern that models how the presence of an existing copy of the element can be checked,
- a $\llcorner\text{failure}\lrcorner$ transition to a *generate*-state,
- a *generate*-state, whose Story Pattern models the actual creation of the copy,

- a transition back to the *check*-state.

Within the *GT* domain, the Story Pattern modeling the *check*-state should contain a traceability link from a $\ll\text{bound}\gg$ node that represents the element from n_{copy} to another node of the same type. Similarly, the derived Story Pattern representing the *generate*-state should model the creation of a traceability link from the $\ll\text{bound}\gg$ representation of n_{copy} to the node representing the copy.

The information that is transferred from an element to its copy is defined by the attributes of that element's type. The type of an element is defined by the element's metaclass and all transitive superclasses of that metaclass.

The embedding of the “*check and generate if needed*” pattern within the existing control flow needs to take into account that the Story Pattern in the *Copy* domain may have outgoing $\ll\text{each-time}\gg$, $\ll\text{success}\gg$ and $\ll\text{failure}\gg$ transitions. The states described above need to be inserted after the $\ll\text{each-time}\gg$ transition in the case of a loop and otherwise after the $\ll\text{success}\gg$ transition. In the case the copy operator is applied within a Story Pattern associated with a $\ll\text{loop}\gg$ state without an outgoing $\ll\text{each-time}\gg$ transition, an explicit $\ll\text{each-time}\gg$ transition from and to the “copy state” needs to be generated first. This ensures that all generated states are executed for each iteration of the loop.

8.2.2.2 Desugaring Composition Path

Each rewrite node that is part of a composition relation needs to be mapped to a “*check and generate if needed*” pattern described above.

Additionally, each outgoing composition relation needs to be mapped to a “select state”. Such a state needs to be associated with a Story Pattern that iterates over all elements from the composition relation.

Finally, the addition of the copies of the contained elements to the copies of the container elements is realized by mapping the composition relation to a $\ll\text{create}\gg$ link with an $\ll\text{onCopy}\gg$ construct on each end. This link is further refined into more operational constructs by the rule from subsection 8.2.2.4.

8.2.2.3 Desugaring $\ll\text{onCopy}\gg$ on Attribute Assignments

In the context of a rewrite node n_{aa} , attribute assignments that are decorated with the $\ll\text{onCopy}\gg$ construct need to be mapped to attribute assignments on a rewrite node representing the copy of n_{aa} . An efficient implementation that satisfies this mapping rule can execute the attribute assignment already at the time the copy of n_{aa} is created.

8.2.2.4 Desugaring $\ll\text{onCopy}\gg$ on $\ll\text{create}\gg/\ll\text{destroy}\gg$ Links

A $\ll\text{create}\gg$ or $\ll\text{destroy}\gg$ link with $\ll\text{onCopy}\gg$ ends that are attached to rewrite nodes n_{src} and n_{trg} needs to be mapped to a link for which the corresponding ends are attached to rewrite nodes representing the copies of n_{src} and n_{trg} respectively. Moreover, the link's side-effect (*create* or *destroy*) needs to be performed in the context of a rewrite rule that is performed when the copies of these nodes are already created.

Remark that the previous chapter defined the $\ll\text{onCopy}\gg$ construct only in the context of $\ll\text{create}\gg$ links. However, since the destruction of links within the target subgraph seems useful and does not complicate the design of the higher order transformation, the mapping rule

is defined for `«destroy»` links too. In contrast, since a combination of the `«onCopy»` construct and the `«destroy»` construct for nodes does require more investigation, that extension is treated as future work.

8.3 Higher Order Transformation in Story Diagrams

This section highlights some key aspects of the higher order transformation model. Since Chapters 5, 6 and 7 already contain detailed descriptions of transformation models in Story Diagrams, the focus of the discussion is shifted from fine-grained explanations of individual Story Patterns to more high-level design considerations.

8.3.1 Story Diagrams

The complete transformation behavior is modeled by seven Story Diagrams. Four of these Story Diagrams relate directly to the mapping rules described above. The other three Story Diagrams realize some small, supportive, control flows.

At the highest level of abstraction, the Story Diagram called “*copy2GT Flow*” iterates over all nodes in the input transformation model that are decorated with the `«copy»` stereotype. After navigating to the related Story Pattern, all Story Diagrams (i.e., decorated activity diagrams) that refer to this Story Pattern are transformed *in-place* into a low-level equivalent. This transformation is modeled by the Story Diagrams discussed below.

At about the same level of abstraction, a Story Diagram called “*Select transition after which new states should be inserted*” models the different kind of transitions that can leave a state within which the copy operator is applied. This Story Diagram formalizes the mapping rules discussed in Section 8.2.2.1.

At the next level of abstraction, a Story Diagram called “*Recursively generate check/generate/manipulate/select pattern*” models how each composition path from a copied node to its contained elements is transformed recursively.

At the lowest level of abstraction, a Story Diagram called “*Generate check/generate/manipulate pattern*” models how the copying of individual elements can be realized. This involves generating states, integrating these states in the existing control flow, and generating Story Patterns that model the behavior of each generated state.

It turns out that one of the three supportive Story Patterns needs to realize some copy behavior too. Consequently, it could be modeled elegantly using the copy operator itself. With other words, it may be desirable to model the behavior of the new copy operator using (another version of) that copy operator. In fact, since the higher order transformation described in this chapter can be considered as the first “compiler” for the Copy Operator, that compiler may now be *bootstrapped* [111, 143].

8.3.2 Story Patterns

The complete transformation is modeled by more than fifty Story Patterns. Some of these patterns realize trivial behavior such as “*check whether a particular loop state has an outgoing «each time» transition*”. Other Story Patterns involve the creation of several tens of nodes and edges. It should be noted that the complex Story Pattern can obviously be refactored into simpler ones by relying on some additional control flow constructs. However, the *Copy2GT*

Figure 8.9: Complete view on the Story Pattern for generating the “check and generate if needed” pattern.

Figure 8.10: View focusing on the only `<<destroy>>` link of the Story Pattern.

transformation has intentionally been designed to evaluate the scalability of Story Pattern specifications.

Figure 8.9 shows all nodes and links from the Story Pattern for generating the “check and generate if needed” states that are described in Section 8.2.2.1 and illustrated in Section 8.2.1.1. The Story Pattern also generates the Story Patterns related to those states. The diagram from Figure 8.9 is not suited for reading, explaining, or changing the rewrite rule at the detailed level. Instead, it does have a useful role. More specifically, the view clearly illustrates the Story Pattern under consideration almost only creates new elements. At the very right of the diagram, the only `<<destroy>>` link from the rewrite rule is shown.

Imagine the Story Pattern contains no additional documentation. In that case, one may want to learn more about this exceptional link. Thus, one would create a new view on the Story Pattern, showing only the source and the target of the `<<destroy>>` link. To understand the role of these elements, one would then add all elements related to the source and target node. These operations are supported by a general purpose UML tool.

Figure 8.10 shows the result of these “reverse engineering” steps. The diagram shows that the link under consideration is supposed to disconnect a transition from the input Story Diagram from its subsequent state. Moreover, the transition, called *afterTransition*, is connected with a node called *checkExistsState*. When combining these observations with some background information (such as the expected behavior of the *Copy2GT* transformation and the name of the complete Story Pattern), one should understand that this view on the rewrite rule models how the newly generated “check” state is being inserted in the control flow of the input transformation model.

The *toState* node represents the previous target of the *afterTransition* node. The diagram shows that a new transition, called *closeCycleTransition*, is set to that target. Again, some background information should enable one to understand that this transition represents the last transition of the generated “*check and generate if needed*” state pattern. These insights can be stored for later reference. More specifically, the engineer that is trying to understand the (hypothetically undocumented) huge transformation rule could store the the view from Figure 8.10 in a diagram called “Embed check/generate state-pattern in transformation flow”. Afterwards, the engineer could apply the same steps to learn more about the nodes *check-ExistsState* and *closeCycleTransition*. This would lead to insights about the relation between these nodes and the node representing the state within which an element is actually copied.

In summary, the definition of the higher order *Copy2GT* transformation as Story Diagrams has been an interesting modeling exercise. The transformation is the most complex application of the UML profile for Story Diagrams at the moment of publishing this thesis. The interested reader is therefore invited to “browse” the actual transformation model, that is publicly available in the MoTMoT project [178].

8.4 Lessons Learned

As the *Copy2GT* transformation is modeled using Story Diagrams, it serves as an additional evaluation of the proposed approach to modeling refinement transformations. Indeed, the case study confirms most of the lessons learned from the *UML2CSP* transformation from Chapter 6 and the *CM2RM* transformation from Chapter 7. However, unlike the transformations from these previous chapters, the *Copy2GT* case study presented in this chapter involves a higher order transformation. Therefore, this section focuses on an additional lesson learned from modeling that kind of a transformation.

“With Great Power there must also come – *great responsibility*” [145]

Model-driven engineering can increase quality and productivity by encapsulating common programming patterns in first-class modeling constructs and mapping these constructs automatically to the implementation by means of model transformations [190]. Indeed, by automating mappings one avoids that human mistakes are made in the process of performing repetitive tasks. On the other hand, the chance that an error in one application actually results to a system failure is much smaller than the chance of failures due to errors in a transformation. More specifically, each application that is generated by an erroneous transformation will contain the related error.

When higher order transformations are used, transformations are generated (or changed) as well. Thus, an error in the higher order transformation propagates to all derived transformations and all applications generated by those transformations. The good news is that as soon as an error is removed from a higher order transformations, all derived transformations and their generated applications can be regenerated automatically. However, although this seems simple in theory, it tends to involve practical problems at the deployment level.

More specifically, the following issues need to be resolved to make the development of higher order transformations more practical with the MoTMoT prototype:

- The existing implementation of the *Copy2GT* transformation needs to be executed explicitly on a specific transformation model, such as *CM2RM*. The output transformation

model can then be transformed into executable code by the MoTMoT code generator. The build system of MoTMoT needs to be revisited such that particular higher order transformations, such as *Copy2GT*, become an integrated part of the “transformation compilation” process.

- Although MoTMoT provides a *Maven* based infrastructure for managing the *compilation*, *testing* and *deployment* and *versioning* of model transformations [248], a systematic process for distributing new versions of a higher order transformation has not been defined yet.

The Maven infrastructure enables a developer of a higher order transformation to distribute a new version to all first order transformation writers automatically, over the Internet. Writers of first order transformations should thus never be bothered with recompiling higher order transformations. In fact, such developers might be unaware of the fact that such transformations are executed behind the scenes.

We have encountered some practical problems with higher order transformations in a distributed development context. For example, I have once distributed a new version of the *Copy2GT* transformation without distributing a new version of a supportive transformation. Obviously, this mistake caused unexpected behavior for remote users of the higher order transformation whereas all tests had passed on my machine. Therefore, one should investigate such practical problems related to “model transformation in the large” too [144]. The different focus relates to the distinction between “programming in the large” versus “programming in the small” [57].

8.5 Summary and Outlook

This chapter formalized the mapping from the *Copy* language to the *GT* language. The latter language corresponds to the UML profile for Story Diagrams defined in Chapter 5 whereas the former language extends that language with the copy operator defined in Chapter 7. Since both the *Copy* and the *GT* language are used for modeling transformations, the *Copy2GT* transformation has the special characteristic of being *higher* (i.e., second) *order*. The effect of the *Copy2GT* transformation is illustrated on the *CM2RM* transformation that has been discussed in Chapter 7. The behavior of the *Copy2GT* transformation is modeled by means of Story Diagrams.

The higher order transformation involves several complex Story Diagrams and Story Patterns containing more than fifty nodes and links. Therefore, the case study is a good evaluation of the proposed techniques for visual transformation modeling. Especially the concept of transformation views turns out to be very useful for managing the complexity of large rewrite rules.

Additionally, this chapter derives some lessons that are specific to higher order transformations. This kind of transformations requires special support for distributed team development. Although the tool prototype that supports this thesis provides a promising infrastructure, more research is needed to manage the transformation deployment process.

Part III

Model Synchronization

This part of the thesis investigates to what extent the proposed transformation modeling approaches support model synchronization. As indicated in the taxonomy from Chapter 2, this transformation type involves the following set of specific requirements:

- *versatile*: ideally, the transformation model is enforceable, checkable, ignorable, and prioritizable;
- *multidirectional*: transformations do not have a unique “source” to “target” direction;
- support for *change propagation*: changes to one model should be reflected in all related models, by triggering minimal modifications to such models.

Chapter 9 evaluates the use of standard OCL and standard Story Diagrams while Chapter 10 considers the potential of using more declarative language constructs from the domain of *Triple Graph Grammars*. The proposed *hybrid* transformation modeling and *reflective* Triple Graph Grammar techniques are interesting extensions to the core contributions of this thesis.

Remark that this part of the thesis describes ongoing work that is not (yet) fully supported by a tool. The models from Chapter 9 do rely on a small prototype based on YATL4MDR [64] and MoTMoT [178]. Moreover, Chapter 10 relies on the design of MOTE [216] and MOF-LON [4]. However, especially the latter chapter is intended to direct the reader to promising *future* work.

Chapter 9

CAViT: Consistency Maintenance with Transformation Contracts

This chapter is based on revisions of the preliminary work from my first EDOC paper [261]. A first revision has been published in the electronic proceedings of a Dagstuhl workshop in 2005 [260]. Another derivative has been published in the proceedings of an ECMDA workshop on traceability in 2006 [257].

The chapter investigates whether it makes sense to complement the *constructive* modeling approach of Story Diagrams with OCL's *restrictive* modeling constructs. As stated in Chapter 5, Stölzel et al. investigated how OCL expressions can be embedded in Story Diagrams at the fine-grained expression level [242]. OCL is used to define the patterns of model elements that need to be transformed. Similarly, the OCL element navigation constructs are embedded in languages such as ATL [23], YATL [204], EOL [134] and (all sub-languages of) QVT [197].

However, one has not yet investigated to what extent the underlying *design by contract* paradigm is useful for modeling transformations. This chapter illustrates how a minimal extension of that paradigm enables one to model transformations that support model synchronization.

The key extension is that so-called “*transformation contracts*” can be maintained automatically by relating invariants to postconditions. When an invariant is violated, the corresponding method will be called provided that its precondition is satisfied.

The *Contract Aware Visual Transformation* (CAViT) framework supports this new paradigm with standard modeling languages:

- transformation contracts are modeled by standard OCL constructs, whereas
- the constructive behavior of transformation methods is modeled using the UML profile for Story Diagrams.

Section 9.1 introduces the reader to the existing design by contract paradigm, Section 9.2 explains how the example transformation from Chapter 7 can be extended with support for model synchronization using the new, “contract aware”, transformation modeling approach and section 9.3 describes the architecture of a tool prototype that supports this approach. Section 9.4 relates this contribution to other work and section 9.5 concludes this chapter.

9.1 Model Synchronization using Design By Contract

This section describes the general design by contract fundamentals in subsection 9.1.1 before introducing a transformation-specific extension in subsection 9.1.2. The new “contract aware transformation modeling” paradigm is a simple extension of the design by contract foundations for maintaining the consistency between related models.

9.1.1 Design by Contract

Design by contract is a software correctness methodology for procedural and object-oriented software. It relies on logical assertions to detect implementation mistakes at run-time or to prove the absence thereof at compile-time.

The fundamentals of design by contract were developed by Floyd and Hoare in the late sixties [87, 115]. By formalizing the effect of programming language constructs on the state of variables in axioms and inference rules, Hoare illustrated the feasibility of proving program correctness. The proposed proof systems are based on *state assertions*, which are logical expressions about the values of program variables. These state assertions are used to state that a program S will ensure a state assertion q (called the *postcondition*) provided that state assertion p (called the *precondition*) holds right before it is executed, or $\{p\}S\{q\}$. A correctness proof consists of a deductive sequence of state assertions and axioms or inference rules from precondition to postcondition.

In the early seventies, Hoare and Wirth published a proof system for the programming language PASCAL [116]. Meanwhile, verification condition generators based on backward substitution were built to automatically derive proof obligations [119]. Finding what rules form the shortest path from precondition to postcondition remains a creative activity but a theorem prover can help discharging proof obligations given a library with the required axioms and inference rules. Today, one can rely on industrial-strength proof assistants with inference rules for object-oriented languages [52].

Design by contract has also found its way into incomplete verification methods. In this context, the precondition and postcondition are checked at test execution time. For this purpose, Meyer included support for expressing assertions in the Eiffel programming language [21] while Kramer built *iContract* for extracting assertions from Java comments [138]. The advantage of the testing approach is that it is applicable even when complete coverage is unfeasible. On the other hand, deviations between the contracts and the implementation may find their way to the production environment due to an incomplete set of test cases.

9.1.2 Contract Aware Model Transformation

In contrast to the transformations from Parts I and II, transformations that support *consistency maintenance* are directly related to consistency constraints. Such constraints express validity rules that need to be maintained “at all times”. In other words, such constraints express what needs to remain constant (or “invariant”) between a set of models.

Therefore, in an object-oriented context, it makes sense to model consistency constraints as *invariants* of special classes. In fact, such classes may provide a set of methods that realize model transformation rules. Remark that in a standard design by contract context, invariants are supposed never to fail. If an invariant does fail at run-time, the system is exposing a programming error. In a consistency maintenance context however, constraints tend to be

violated frequently. Of course, special actions are supposed to react on such violations and take the appropriate measures to bring the system back into a consistent state.

Therefore, from a modeling perspective, one needs a means to associate transformations with consistency constraints. Coming back to object-oriented transformation modeling constructs, one needs a means to associate a method with an invariant. More specifically, one needs to express which method maintains which invariant. Since the effect of a method is formalized by its postcondition, one can model the desired dependency by relating postconditions to invariants.

Obviously, invariants may be violated in a wide variety of scenario's. Therefore, methods that would need to maintain invariants *in general* would become undesirably complex. Consequently, one would like to express that a method only maintains a particular invariant *under well-defined conditions*. In a design by contract context, the precondition of a method is a natural means to model the limitations of a such methods.

In summary, one can define the concept of a transformation contract as follows:

A transformation contract is a pair of constraints (called the pre- and post-condition) that describe the effect of a method on a set of models. The postcondition describes the effect of the method whereas the precondition describes when the method can be executed successfully.

This definition does not yet extend the design by contract fundamentals. However, in a model synchronization context, the basic concepts need to be complemented with a small extension:

Within a transformation class, the postcondition of a method can be related to an invariant of that class. In that case, the method is supposed to maintain the invariant in all cases that its precondition is satisfied.

For simple contracts, one can automatically generate constructive behavior from a restrictive specification (such as an invariant or postcondition). In fact, the transformation contracts that are modeled implicitly by the OCL expressions within ATL [127] rules have as a precondition that the target elements do not exist yet. This simplifies the contracts significantly, since this eliminates the need for change propagation techniques.

On the other hand, complex cases require that the actual behavior of the method is modeled explicitly. More specifically, it may occur that a desired user interaction scenario is completely transformation specific. In that case, algorithms that automatically generate behavioral transformation models cannot be used.

Section 9.1.2.1 describes how transformation contracts can be used to model consistency maintenance systems whereas Section 9.1.2.2 describes the relations to inconsistency management.

9.1.2.1 Consistency Maintenance

Given the extension described above, the notion of transformation contracts can be used for modeling transformation that support *consistency maintenance*:

A transformation contract of a method can be *maintained automatically* by calling the method (1) as soon as the invariant corresponding to its postcondition is violated and (2) provided that its precondition is satisfied.

Note that several methods can have the same postcondition. In the case this postcondition is mapped to an invariant, several methods become a candidate for maintaining the consistency constraint that is modeled by that invariant. However, not all preconditions of these methods may be satisfied. In the special case where the precondition of more than one of the candidate methods is satisfied, the choice of the method to be called is non-deterministic. An engine could delegate the choice of such a method to the modeler. From the perspective of the user of a consistency maintenance system, this corresponds to the approach proposed by Becker [17].

9.1.2.2 Inconsistency Management

Some model inconsistencies cannot be resolved automatically. More specifically, some invariants may not be mapped to a postcondition. Even when an invariant is mapped to several methods, it may occur that neither of these methods has a satisfied precondition. Formally, this may occur when the logical *or* of these methods' preconditions does not imply *true*. When such an inconsistency occurs, a transformation engine should report the inconsistency to the modeler.

9.2 Example Transformation Models

This section provides a detailed description of two example transformation methods.

The transformation under consideration extends the one from Chapter 7. More specifically, it covers the *synchronization* of classes in a conceptual model with entities in a robustness model.

The consistency of conceptual and robustness models can be maintained by a combination of transformation contracts. Note that in the case of multiple contracts with the same postcondition, it may be desirable to make their preconditions mutually exclusive. This ensures that only one method is called in a particular inconsistency scenario.

As a first example of a consistency contract, consider the following OCL fragment:

```

1 context CMconsistentRM:
...

42 let CMconsistentRMcontract(): Boolean=
43   conceptualModelTracesToRobustnessmodel() and
44   allClassesFromModel(cm)->forall(cc: Classifier |
45     allClassesFromModel(rm)->exists(rc: Classifier |
46       this.traces->exists(t2 |
47         t2.node->exists(cNode | cNode.content=cc)
48         and t2.node->includes(rNode | rNode.content=rc)
49       ) and
50       cc.name=rc.name and
51       rc.hasStereotype("entity") and
52       cc.attributes()->forall(ca| -- cc and rc have 'same' attributes
53         rc.attributes()->exists(ra| -- (note: attributes() defined on Class)
54           ca.name=ra.name and
55           ca.type.name=ra.type.name and (-- represent 'same' type
56             ca.type<>ra.type or ( -- different class because should be copied
57               -- take care of potentially used built-in types etc.
58               not allClassesFromModel(cm)->includes(ra.type)
59             )
60           )
61         )
62       )
63     )
64   )

```

...

```

73 inv:
74   CMconsistentRMcontract()

```

The *CMconsistentRMcontract* helper operation formalizes the core of the mapping between conceptual and robustness models.

In order to maintain this rather complex consistency constraint, a transformation system should handle the case where only the conceptual model exists and the robustness model still needs to be generated (case 1) as well as the case where the robustness model has been generated before (case 2). In the latter case, the transformation may have to search for all classes unrelated to an entity and either generate the entity automatically or allow the user to link the class to an existing entity manually. Chapter 7 discusses in detail how to model case 1 in a compact manner. Case 2 will be handled by this chapter. Remark that the contract for both cases is defined in *standard* OCL, which is in contrast with the previous approaches described in Section 1.3.

The following transformation contract formalizes how the *cmClasses2rmEntities* transformation method presented in Chapter 7 supports this consistency constraint:

```

76 context CMconsistentRM::cmClasses2rmEntities(): boolean
77 pre:
78   not conceptualmodelTracesToRobustnessmodel()
79 post:
80   CMconsistentRMcontract()

```

In order to manage the synchronization of classes and entities in existing models, the rather complex *CMconsistentRMcontract* constraint is decomposed into a number of more fine-grained constraints. It should be noted that this decomposition not necessarily improves the understandability of the overall effect of the consistency contract. However, it does enable one to distribute the related transformation responsibilities across several transformation methods:

```

116 context CMconsistentRM::fix_eachClassTracesToAnEntity_violated_rmExists(): boolean
117 pre:
118   conceptualmodelTracesToRobustnessmodel() and -- 'rm' not Undefined
119   not eachClassTracesToAnEntity()
120
121 post: eachClassTracesToAnEntity()
122
123 context CMconsistentRM::fix_ClassEntity_name_violated(): boolean
124 pre:
125   not classEntity_name_match()
126 post:
127   classEntity_name_match()

```

This OCL fragment relies on some OCL helper operations, including *eachClassTracesToAnEntity*. The definition of most helpers is left out of this text due to presentation considerations. However, as an example, the *eachClassTracesToAnEntity* helper is defined below.

It should be noted that the helper operations rely on one another. Line 92, for example, calls the *conceptualmodelTracesToRobustnessmodel* helper to ensure that the robustness model has been generated before. Such dependencies ensure that the different transformations are scheduled in the right order: it does not make sense to start generating new entities if the model in which they need to be stored has not been generated yet.

```

91 let eachClassTracesToAnEntity(): Boolean=
92   conceptualmodelTracesToRobustnessmodel() and -- 'rm' not Undefined

```

Figure 9.1: Story Diagram for a transformation that establishes the *eachClassTracesToAnEntity* constraint on existing models.

```

93 | allClassesFromModel(cm)->forall(cmc|
94 |   allClassesFromModel(rm)->exists(rmc|
95 |     this.traceabilityLinks->select(oclIsKindOf(Class2Entity))->exists(l|
96 |       l.node->contains(cmc) and
97 |       l.node->contains(rmc)
98 |     )
99 |   )
100 | )

```

Figure 9.1 models the behavior of the method called “fix_eachClassTracesToAnEntity_violated_rmExists”. The contract of this method is shown from line 116 to line 121 of the OCL fragment shown above. The contract formalizes that the method under consideration ensures that the classes and entities within two existing models remain synchronized.

As can be seen on Figure 9.1, the transformation method “fix_eachClassTracesToAnEntity_violated_rmExists” iteratively looks for violating classes in the conceptual model by iterating over all classes in the conceptual model. For every match, the transformation checks whether the matched class is related to an entity.

If the latter “Is the Class related to an Entity?” pattern does *not* match, the transformation has found a violating class: the transition with the *<< failure >>* stereotype is triggered and the code state containing a *setFocus* call highlights this problem such that the developer can solve it manually [257].

If the “Is the Class related to an Entity?” pattern does match, the transition with the *<< success >>* stereotype is triggered and the transformation continues with the next class in the conceptual model. After visiting all such classes, the transformation returns *true* if it has established its postcondition, which corresponds to the *eachClassTracesToAnEntity* constraint, or resumes the iteration over violating classes.

Figure 9.2 shows how the behavior of the second state can be modeled as a Story Pattern. The pattern can be read as starting from the bound *applicationModel* node. It searches for the package *wodnApplication* containing the robustness model *rm*. The *rm* node should be connected to a stereotype that has been bound in the first state of the transformation. The *c2e* node represents a traceability link between an entity in *rm* and the class that was bound in the

Figure 9.2: Story pattern modeling the transformation behavior in state “Is the Class related to an Entity?”.

Figure 9.3: MOF instance defining the structure of the *CMconsistentRM* transformation and its traceability links.

`<< loop >>` state called “For all classes in the CM”.

As a second sample consistency constraint, consider *classEntity_name_match*. This OCL helper operation checks for all elements satisfying *eachClassTracesToAnEntity* whether the names of a class and its related entity correspond.

```

102 -- Evaluate whether the classifiers (including entities) of all nodes related by
103 -- a Class2Entity link have the same name
104 let classEntity_name_match(): Boolean=
105   traceabilityLinks->select(oclIsKindOf(Class2Entity))->forall(l|
106     -- for all relevant links:
107     l.ignoreConstraints or -- user has marked that the name can be ignored, or
108     l.node->forall(n1,n2| -- for any couple of nodes,
109       -- their content elements should have the same name
110       n1.content.name=n2.content.name
111     )
112   )

```

Note that this constraint implements the fundamental model synchronization concept of “tolerated inconsistencies” as described by Balzer [13]. More specifically, on line 107 it specifies that no further checking is required if the user has set the *ignoreConstraints* property of the *Class2Entity* link to true.

Figure 9.3 shows how the *ignoreConstraints* property is defined on the *Link* metaclass. By exposing this property to developers, a consistency maintenance system allows them to

Figure 9.4: Architectural diagram of CAViT.

postpone the resolution of particular inconsistencies. To evaluate this concept, Frank Altheide has developed a prototype of a UML tool plugin that can highlight model elements on demand and that provides query and update facilities based on the *ignoreConstraints* property [257].

9.3 Architecture of the CAViT Framework

This section briefly illustrates how CAViT is related to existing model management software. Figure 9.4 shows that CAViT acts as middleware between an OCL based consistency checker (YATL4MDR [64]) and the compiler of Story Diagrams discussed before (MoTMoT [178]).

YATL4MDR and MoTMoT can access the model repository through a file-based or object-based interface (XMI [196] or JMI [168]). CAViT lets them use the latter to eliminate expensive serialization calls.

The interface between MoTMoT and CAViT is defined by the attributes and links of transformation objects: these elements are available to the invariant/postcondition definitions as well as to the transformation methods. Behind the scenes, a MoTMoT transformation object implements JMI based interfaces as well. Therefore, it can be treated as a regular model element by the general purpose OCL evaluation engine.

9.4 Related Work

The CAViT framework bridges two technological spaces. *Object-oriented metamodeling* is used to define the structure and contracts of transformation definitions. The behavior of transformation methods is modeled by controlled graph rewriting rules. Controlled graph rewriting is a well-known concept in the *graph transformation* technological space. By formalizing the technology bridge with mainstream *design by contract* concepts, this chapter is related to both technological spaces. However, since related work from the graph transformation space has already been discussed rather extensively in Sections 1.3.4 and 7.4, the discussion in this section is quite focused.

9.4.1 Object-Oriented Metamodeling and Consistency Maintenance

CAViT is obviously not the only framework for modeling transformations in an object-oriented manner. As described in Chapter 1, Akehurst proposed to model transformation definitions as mathematical relations [2] using an object-oriented library. A relation maps one metaclass to another one and is defined by:

- a domain and a range specification that states which instances of the source and target metaclass can be mapped onto each-other,
- set-theoretical properties such as *bijective*, *functional*, *total*, etc., and
- relation-specific constraints that consist mainly of equations between attributes of related elements.

All three constraint types are modeled as invariants on a *Relation* class. The second class of constraints is made available automatically to all relations by means of inheritance (from the *Relation* class described in the appendices of [2]). Since in CAViT all consistency constraints are modeled as invariants on classes as well, it is compatible with this framework for model transformation based on set theory.

Reasoning about one source and one target model is a special case of the CAViT approach where transformation definitions can hold one, two, or more attribute references to models. Apart from this, the added value of CAViT lies in the definition of behavior for constraint violations: CAViT will delegate to the model transformation method of which the postcondition is mapped to the failed invariant. By reasoning about invariants, pre- and postconditions, we build upon the well-established foundations of design by contract.

Moreover, since CAViT is based on Story Diagrams, it provides a UML based syntax to describe the behavior of transformation methods. The latter is a natural extension of the *relational* approach since Akehurst only implemented the bodies of transformation methods in Java due to the absence of precise UML action semantics [3]. Another possibility is the extension of OCL as a side-effect free constraint and query language to a transformation language. Initial experiments were already conducted to build consistent model elements automatically for architectural models [239]. More case studies are required to assess the readability and expressiveness of Story Diagrams and the OCL action language in practice.

In general, the MDE community lack case study solutions involving model synchronization. Kermeta, for example, extends the MOF with an OCL based transformation language [179]. Similar to CAViT, Kermeta supports the realization of transformation definitions as classes and transformation rules as methods. Kermeta is also compliant with OMG standards where possible. However, unlike CAViT, Kermeta is not designed to model transformations supporting consistency maintenance. Moreover, it is unclear why Kermeta would support the cognitive dimensions presented in Section 2.4.2 better than a Java based solution, as presented in Section 1.2.3.2.

It should be noted that work in the object-oriented metamodeling space builds upon the results of the logic-based knowledge bases. In the latter area, Balzer was among the pioneers that proposed to decouple the definition of restrictive consistency contracts from constructive repair actions. Moreover, he recognized the importance of temporarily tolerating consistency violations and the role of manual reconciliation assisted by automatic inconsistency notification [13]. Finkelstein et al. used executable temporal logic [174] to implement transformation rules for maintaining the consistency between software models from different viewpoints [85].

9.4.2 Graph Transformation

As discussed in Section 1.3.4, the roots of the graph transformation space were already developed in the early seventies [113]. The foundations of controlled graph rewriting are described by Schürr in first volume of the handbook of graph transformation [230]. In fact, the second volume already refers to the support for design by contract within Progres [227]. In contrast to the transformation contracts defined in this chapter, the Progres approach is procedural instead of object-oriented.

Other authors have proposed the use of preconditions, postconditions and invariants in the graph transformation space too [209]. Based on these concepts, tools can analyze whether a rule does not break the well-formedness rules of a modeling language [247]. Tom Mens mapped Prolog-based work on “reuse contracts” to the graph transformation space to illustrate the applicability of graph transformation to the evolution of object-oriented software [249]. Reuse contracts are model transformation rules that were defined to manage the evolution of class hierarchies and collaborations. By defining a set of primitive reuse contracts as graph rewrite rules, Mens was able to derive a conflict matrix. This idea was later elaborated in collaboration with Tüntzer and Runge [162]. One should note that the algorithms for computing conflict matrices have not been designed with the control structures and the copy operator introduced by this thesis in mind.

Since our definition of a transformation contract is not based on a particular constraint language such as the OCL, CAViT’s consistency maintenance approach should be applicable to visual specifications of pre- and postconditions as well. This chapter complements previous work on pre- and postconditions within the graph transformation space since one only investigated how inconsistency could be prohibited, rather than being tolerated temporarily. Once the work of Mens et al. would be extended to support controlled graph rewriting, it could be used to compute possibly undesired side-effects of executing one transformation method before another one, when CAViT indicated that both of them could fix a model inconsistency.

9.5 Summary and Outlook

The main theoretical contribution of this chapter is that it relates new developments in model transformation technology to existing paradigms. More specifically:

- transformation definitions and transformation rules can be realized using classes and methods,
- one or more models from a transformation definition can be made accessible in a transformation class by means of references to elements from these models,
- consistency contracts can be realized using class invariants,
- the postcondition of particular methods should be related to these class invariants, and finally
- a violated invariant can be corrected automatically by calling the method whose postcondition is related to that invariant and whose precondition is satisfied.

Additionally, the chapter relied on object-oriented traceability metamodels too: a transformation class can be associated with a hierarchy of traceability classes. This design has also been

useful in the context of Chapters 6 and 7. Finally, the behavior of the transformation methods can still be modeled by Story Diagrams. In summary, even transformations that support model synchronization can be modeled using standard, object-oriented languages.

Coming back to the challenges that were identified in Chapter 3, one should keep in mind that several OCL related transformation problems still remain to be solved. As a promising direction for future work, one may investigate whether contract-aware model transformation is applicable for composing primitive refactoring operations in the style of Section 3.3.1 too.

Chapter 10

Towards Hybrid Transformation Modeling

This chapter is based on a technical report that I have presented on the first international workshop on Triple Graph Grammars [264] in 2006.

As discussed in Section 2.4.1.4 from the taxonomy chapter, transformation languages can be divided in two major categories: on the one hand, restrictive languages only support the specification of constraints that need to be maintained by a transformation. On the other hand, constructive languages also support the specification of the modifications that need to be applied for establishing these constraints on a set of models. This classification is often wrongly associated with the distinction between *declarative* and *operational* model transformation languages.

The root of that confusion may be that the concept of a “declarative specification” is severely overloaded in a general purpose programming context. The taxonomy from Chapter 2 aims to avoid this pitfall by precisely defining the features related to declarative transformation modeling. More specifically, a transformation model is said to be *fully declarative* when it is implicit with regards to *execution direction*, *change propagation* and *rule scheduling*.

This chapter illustrates that some restrictive languages do not enforce a fully declarative transformation modeling style. Moreover, this chapter presents transformation models in a language with some operational features applied locally. However, the example transformation rules still satisfy *all* properties of fully declarative transformation models. Therefore, the transformation modeling style is called *hybrid*. The language is based on rule-based graph rewriting (Triple Graph Grammars) and controlled graph rewriting (Story Diagrams).

Finally, the chapter briefly refers to a related approach that supports the refinement of restrictive transformation models that are fully declarative into constructive transformation models that are operational. When properly managing the related higher order transformation, transformations can be modeled restrictively at a high level of abstraction while constructive details are added at a lower level of abstraction.

Although this final chapter of the thesis may seem purely language theoretical at first sight, its goal is quite practical: by illustrating that the combination of language features may lead to more intuitive transformation models, we aim to avoid that transformation languages become overly “pure”. This concludes the thesis with an outlook for future work: the primary require-

ment for a transformation modeling language should be that it minimizes the gap between the mental model and the models used for presentation and execution purposes.

10.1 The Need for Hybrid Transformation Models

In today's realization of OMG's Model Driven Architecture (MDA [192]) vision, a main research challenge remains the development of languages that facilitate the specification of complex systems in an intuitive manner. This holds for application modeling languages as well as for transformation modeling languages.

Although the QVT standard refers to the combination of its “declarative” and “operational” sub-languages as a “hybrid” language, it does not contain a concrete case study to motivate the need and to clarify the nature of a hybrid transformation language. The notion of a “hybrid” language has been coined in the context of the Atlas Transformation Language (ATL [127]) too but the proposed language is only “declarative” with respect to the an aspect of rule scheduling.

In this chapter, we use MDA modeling techniques for improving the language support for the development of model transformations. More specifically, we present the use of a new, *hybrid*, transformation language in the context of the case study from Chapters 7 and 9. By limiting the size of the case study, we are able to focus on the issues that challenge today's state-of-the-art transformation tools. One particular challenge is *developer interaction*: tools supporting declarative languages are usually limited to batch-transformations or only support a fixed interaction pattern [17]. Tools supporting operational transformations can be used for implementing any kind of interaction scenario but the transformation models tend to be rather low-level [257]. By implementing the hybrid language as a UML profile, tool support can be realized with reasonable effort.

This chapter is organized as follows: Section 10.2 presents an operational and a declarative graph transformation solution to the presented case study. Based on the problems identified in Section 10.2, Section 10.3 covers the main contribution of this chapter by introducing a concrete application of a new, *hybrid*, transformation language. After pointing to related work in Section 10.4, the chapter concludes.

10.2 Balancing between Operational and Declarative

The case study from the previous chapter challenged our use of these operational languages (OCL and Story Diagrams) in that bidirectional consistency constraints could not be modeled concisely. A bidirectional consistency constraint can be maintained by a pair of Story Diagram transformations but this is undesirable due to the verbose specification style.

Interestingly, the case study also challenges a fully declarative formalism, such as Triple Graph Grammars (TGGs), on various aspects. For example, the implementation of a transformation that supported a realistic developer interaction process required us to add control structures between TGG rules (or implement them on a lower level of abstraction [259]). Subsections 10.2.1 and 10.2.2 present these challenges in more detail.

10.2.1 Story Diagrams with OCL: too low-level

Coming back to the combined use of OCL and Story Diagrams, it should be emphasized that the two example transformation contracts presented in the previous chapter formalize only a small fragment of the complete relation between conceptual and robustness models. To illustrate that more contracts are needed, consider the relation between the attributes from a class in a conceptual model and the attributes from the related entity. Not only should the *name* property of these attributes be synchronized, the same holds for the *visibility* and any other property of UML *Attribute* nodes.

Thus, several other contracts such as the one shown below need to be maintained too:

```

311 context CMconsistentRM
312 inv: attr2attr_visibility_match()
313
314 context CMconsistentRM::fix_attr2attr_visibilityMatch_violated(): boolean
315 pre:
316   not attr2attr_visibility_match()
317 post:
318   attr2attr_visibility_match()
319
320 let attr2attr_visibility_match(): Boolean=
321   traceabilityLinks->select (oclIsKindOf (Attribute2Attribute)).forall (l|
322     l.node.forAll (n1,n2|
323       n1.content.visibility=n2.content.visibility
324     )
325   )

```

Moreover, from methods such as *fix_attr2attr_visibilityMatch_violated*, it is expected that all pairs of related attributes with a different visibility are updated, according to one of the following policies:

- update the visibility of the attribute from the class automatically, or
- update the visibility of the attribute from the entity automatically, or
- ask the user to give his explicit permission to ignore the inconsistency, or
- ask the user to change the visibility of one of the attributes manually.

Because this kind of violations can occur for all elements that are constrained in two directions, the transformation contracts should not be modeled explicitly by the transformation writer. Instead, they should be modeled implicitly by special constructs of the transformation modeling language. Otherwise, the low level of abstraction leads to verbose transformation specifications.

10.2.2 Triple Graph Grammars: too generic

Triple Graph Grammars are a natural alternative for Story Diagrams when multidirectional constraints need to be maintained with change propagation. This subsection illustrates how the constraint defined on conceptual and robustness models can be maintained by a set of TGG rules. We will then identify where the TGG formalism needs to be extended to complete the case study in a satisfactory manner.

10.2.2.1 Declarative Transformations: Graph Grammars

A *graph grammar* is defined as a set of rules describing local changes of graphs [221]. These rules are executed “freely” until a fixed point is reached. Graph grammars are a declarative formalism since they do not specify a state-based modification of one graph into another one. In terms of the taxonomy from Chapter 2, the rule scheduling algorithm is *implicit*.

Pair Graph Grammars were introduced in the early seventies to specify graph-to-graph translations. A pair grammar thus consists of rules which modify two participating graphs and update correspondences between nodes from these two graphs.

Triple Graph Grammars (TGGs) were introduced in the early nineties as a formalism for maintaining *bidirectional* consistency constraints between models originating from different software engineering tools [229]. While pair grammars were already successfully applied for the integration of software engineering tools [78], triple graph grammars allowed one to model a *bidirectional* translation system in a more compact manner. A triple rule not only consists of a left- and a right-hand side. Additionally, it divides the rewrite nodes and links in three domains: two domains represent the models that need to be kept consistent. A third domain represents the traceability model. Essentially, a triple rule describes the relations that need to hold between elements from the right-hand side when the elements from the left-hand side are already consistent. At the root of the grammar, an axiom rule defines the consistency between two root elements from the mapped models. Usually, such a rule defines the consistency relation between “package” elements while other triple rules define the consistency of the contained elements recursively.

In the last decade, triple graph grammars were extended to support the disambiguation of conflicting rules by users [18], they were generalized for the integration of more than two models [135] and support for incremental change propagation was investigated more thoroughly [245].

Although TGG rules can be executed directly by a Java interpreter [132], their operational semantics is usually clarified by presenting the mapping of a TGG rule to conventional rewrite rules [136]. Burmester et al., for instance, map TGG rules to six primitive graph rewriting rules [245]: three rules for adapting changes to the source model and three for adapting changes to the target model.

To indicate the direction of the change propagation, the former three are called the left-to-right or “forward” rules while the latter are called right-to-left or “backward”. Both groups of rules consist of a *creation rule*, a *deletion rule* and a *consistency rule*. The former makes sure that when an element is found in one model, a corresponding element exists in the other model. The second rule makes sure that when an element is deleted from one model, its corresponding element is deleted from the other model. The latter rule makes sure that when attribute updates on an element in one model trigger a violation of a consistency constraint related to an element in the other model, that the element in the other model is updated. In fact, an additional rule is needed to create traceability links between model elements that are consistent with one another but were not mapped to one another yet [229].

10.2.2.2 Application to Example

This subsection presents four rules from a triple graph grammar designed for maintaining the consistency between the models from the case study related to conceptual and robustness models. The rule is defined in the context of the *CMconsistentRM* class, which has already

Figure 10.1: TGG Rule ensuring that for each node representing a conceptual model, there exists a node representing a corresponding robustness model and vice versa.

been introduced in the previous chapter.

The rule expresses that an application that is referenced by the *applicationName* property of the *CMconsistentRM* class should contain a conceptual model *cm* and a robustness model *rm*. These nodes of type *Model* should be linked to one another and carry stereotypes for marking a conceptual and a robustness model respectively. The name of the robustness model should be equal to the name of the conceptual model suffixed by “_RM”.

Note that in contrast to the rewrite rules from the previous subsection, the rule from Figure 10.1 is not embedded in an operational control flow. The only bound node is displayed in the top left corner. From this context, all other nodes are matched. The *applicationModel* node can be matched directly, by traversing the outgoing *applicationModel* link from the node called *this*. The *applicationModel* node represents the UML model that contains both the *wodnApplication* package with the conceptual and robustness models such as the ones on Figure 7.1 and 7.2 as well as the package containing the profile definitions for conceptual and robustness modeling.

More specifically, it could be matched to the *n3* node shown on Figure 7.3 when the *this* node of Figure 10.1 would have been initialized with the *n1* node of Figure 7.3. Based on the *closure* stereotype on the *ownedElement* link that leaves the *applicationModel* node, the rule on Figure 10.1 can match the *wodnApplication* node against the *n3* node in the example host graph from Figure 7.3 even though node *n2* is located between node *n1* to *n3* on their connecting *ownedElement* path.

The nodes and edges that are decorated with the *create* stereotype are part of the right-hand side (RHS) of the TGG rule. They are displayed in green and can be divided in four groups, based on whether they carry one of the *left*, *map* or *right* stereotypes. Elements that do not carry any of the three stereotypes are part of the overall host graph. Elements carrying the *left* or *right* stereotypes are part of sub-graphs representing the two models that need to be kept consistent. Elements carrying the *map* stereotype are part of the interconnection (sub)graph (or “traceability model”). They are displayed by a hexagon symbol.

As an illustration that the semantics of a TGG rule is more declarative than a conventional rewrite rule, consider the semantics of Figure 10.1. With conventional rewrite semantics (or without taking the *left*, *map* and *right* stereotypes into account), three new nodes would be created after finding the match described above (i.e., *cm*, *m2m* and *rm*).

Figure 10.2: TGG rule defining the consistency constraint between packages in conceptual and robustness models. Another rule, dealing with packages recursively contained in *cm* and *rm*, is left out of this chapter due to space considerations.

No more checking would be performed afterwards. With TGG semantics however, the rule will create a consistent *rm* node when only the *cm* node is available. Vice versa: a new *cm* node can be created from an existing *rm* node. When both the *cm* and *rm* nodes exist, but they are in conflict, the TGG rule will use the path over the *m2m* node to navigate between conflicting nodes in order to make them consistent again. This can involve changing the name of the *cm* or *rm* nodes or changing the set of stereotypes attached to these nodes. Finally, when a pair of consistent *cm* and *rm* nodes exist without a path over *m2m* connecting them, the TGG rule will create such a path.

The rule on Figure 10.2 specifies that for any consistent triple of (*cm*, *m2m*, *rm*) nodes, all the contained packages should form consistent triples too. Again, consistency is defined as having a counterpart in the other sub-model while that counterpart has the same name. The rule on Figure 10.3 specifies that at all times, classes contained in a package from the robustness model should be mapped to classes with the same name in a corresponding package in the robustness model. Additionally, the classes in the robustness model should be decorated with the *<< entity >>* stereotype.

Note that the three TGG rules presented so far refer to particular nodes representing stereotypes. Figure 10.4 illustrates how the matching of these nodes can be constrained. Figure 10.4 should be considered as a view that complements the three presented TGG rules. The final TGG rule presented here ensures that the types of corresponding attributes are consistent. The rule illustrates the issue of tracking the creation of edges. Since the underlying graph formalism does not support edges pointing to edges, the *at2at* node points to the (*ca*, *cat*) and (*ra*, *rat*) nodes respectively instead of pointing to the RHS edge between these nodes. Note that the management of the Link classes by the *CMconsistentRM* class is implemented in another view

Figure 10.3: TGG rule for classes and entities.

Figure 10.4: View defining how stereotype nodes should be matched in the TGG rules.

that is left out due to space considerations.

Once more, note that the presented TGG rules are not embedded in a control flow and are thus assumed to operate in parallel. Ambiguities can be circumvented by ensuring that the left-hand side of each rule is logically exclusive with that of the other TGG rules, or resolved by offering users the opportunity to order the set of matched rules based on some predefined characteristics. The latter strategy is supported by Fujaba's *MoTE/MoRTEn* plugins and is implemented by flagging elements as soon as they have participated in the execution of one TGG rule such that other rules can be disabled for those elements [216].

10.2.2.3 Problem Identification

The TGG rules presented in the previous section only define the general consistency constraints that should be maintained across conceptual and robustness models. They do not take into account that, for example, the types of attributes can be external datatypes or classes imported from a third party “library” model. The details of this mapping issue are elaborated in Section 7.5.1 from Chapter 7.

Moreover, the semantics of all rules is automatically the same. This implies that, for example, all inconsistencies are resolved fully automatically in both directions. However, it may be desirable that some rules interact with developers before modifying any model element. Moreover, some inconsistencies should be resolved manually, or they should even be tolerated [84]. Nuseibeh even argues that each inconsistency must be treated differently [189].

Figure 10.5: TGG rule for attribute types in the context of mapped classes and entities.

Without going that far, we acknowledge the need for control flow, user interaction and inconsistency tolerance.

A second problem with the presented TGG rules is that they only constrain the mapped nodes based on their name property. Other properties, such as the visibility, and all other properties defined on the *UmlClass* metaclass, should be kept consistent too. Obviously, one could enumerate these properties in the TGG rules explicitly but a more declarative approach is desirable for readability and evolvability reasons.

10.3 A Hybrid Model Transformation Language

This section learns from the problems identified in the declarative and operational approaches to derive a hybrid solution that allows one to apply the best features of both paradigms together.

10.3.1 Control Flow

A simple extension of the discussed TGG system is the addition of a rule that defines how external attribute types should be kept consistent. Figure 10.6 illustrates that such a rule does not introduce any new concepts. The major challenge however is the integration of the TGG rules shown on Figure 10.5 and 10.6. This subsection first discusses a new approach to tackle this problem. It consists of a control flow layer on top of TGG rules. Then, an alternative approach based on a layered transformation specification is presented.

In order to support inconsistencies in a controlled manner, the previous chapter introduced the *ignoreConstraints* property of the *Link* metaclass shown on Figure 9.3. By exposing this property to developers, a consistency maintenance system allows them to postpone the resolution of particular inconsistencies.

Recall the rule from figure 10.3 stated that the names of classes in the conceptual model should be the same as the names of their corresponding entities in the robustness model. Developer should be able to give permission to ignore that constraint for a pair of mapped classes with different names. A user interface should then remove this pair from a list of conflicts to be resolved. For implementing this kind of system, the rule from figure 10.3 needs to be extended

Figure 10.6: TGG rule for handling external attribute types.

by adding “*ll this.ignoreConstraints*” to the constraint “*classInCM.name=entityInRM.name*”. Since tolerated inconsistencies are thus tracked explicitly, each decision can be revisited in later process cycles.

Clearly, the developer interaction methods discussed in the previous chapter can be useful to make TGG systems more interactive too. Therefore, conventional Story Diagram rewrite rules should be available for query purposes while method calls should be available for triggering user interactions.

10.3.1.1 Controlled Triple Graph Grammars

In many cases, part of the control flow can be pulled out of rules by organizing them in a fine-grained manner [257], with logically exclusive preconditions. However, since the two rules from Figures 10.5 and 10.6 can establish consistency under overlapping preconditions, requesting information from the modeler is essential.

Both a branch and a TGG rule call is needed to delegate to the appropriate rule. After identifying the possible cases in which inconsistencies between attribute types can occur, the following paragraph will focus on the overlapping between the application conditions of the two TGG rules.

Both the rule on Figure 10.5 and that on Figure 10.6 can create a consistent type for *ca* (or *ra*) if this type is *null* while the type of the corresponding *ra* (or *ca*) exists already. In fact, several other cases are handled gracefully too. However, the challenging cases from a model reconciliation viewpoint are the ones where *ca* has a type in the conceptual model *cm* and *ra* node has an external library type, or vice versa: in these cases, both the rule from Figure 10.5 and that from Figure 10.6 would match. Adding an additional application condition in both rules is not a feasible solution since input from the modeler is required to resolve this

Figure 10.7: Control flow of the interactive, hybrid, model transformation for reconciling attribute types in the case of one internal type and one external one.

Figure 10.8: Story Pattern for the first state of the transformation from Figure 10.7.

ambiguity.

Instead, an explicit control flow needs to be specified between the two TGG rules. More specifically, before executing the TGG rules, the consistency system needs to look up what attribute type pairs consist of one internal and one external type. For such pairs, the system needs to ask the modeler what type gets precedence over the other one. Figure 10.7 displays such a control flow specification. In the first state, all pairs of linked attributes from the conceptual and robustness model are matched. Figure 10.8 shows the Story Pattern that formalizes when such a match should be found.

As in previous diagrams, the *ca* node represents an attribute from the conceptual model whereas *ra* represents an attribute from the robustness model. The relation between classes and attributes is established by means of the “feature” links from *cc* and *re* to *ca* and *ra*. Intuitively, *cc* represents a class in the conceptual model *cm* whereas *re* represents an entity in the robustness model *rm*. The *stereotypeOnCM* and *stereotypeOnRM* nodes represent the stereotypes with name “Conceptual Model” and “Robustness Model” respectively.

The second and third states from the top of Figure 10.7 test the precondition of this in-

Figure 10.9: TGG rule for external types, refactored to be callable with already bound nodes as arguments.

Figure 10.10: Story pattern modeling the first state of the transformation from Figure 10.9.

teractive transformation: does *ca* have an internal type while *ra* has an external one or vice versa? The second state tests that the type of neither *ca* nor *ra* is null since those cases are handled by the conventional, uncontrolled, behavior of the two TGG rules.

The third state from the top tests whether the type of *ca* is contained in the conceptual model too while the type of *ra* is not contained in the robustness model or, vice versa, that the type of *ra* is contained in the robustness model too while the type of *ca* is not contained in the conceptual model.

The fourth state from the top contains a call to request information from the modeler. Without this human input, the transformation cannot decide whether to change the external type to an internal one or vice versa. The two subsequent states contain a call to *variants* of the presented TGG rules. These rules modify the type of *ca* or *ra*, which resolves the inconsistency.

Calling conventional TGG rules directly is not desirable since they do not operate within the context of particular model element tuples. Therefore, fully declarative TGG rules need to be refactored into more operational ones. As an example, the TGG rule for external attribute types (presented on Figure 10.6) is transformed into the two-state transformation presented on Figure 10.9. Note that these rather obvious Story Diagrams could be generated by a rather simple higher order transformation.

The first state matches all attribute pairs that are linked by an instance of *Attribute2Attribute*. As Figure 10.10 indicates, *ca* and *ra* can be interchanged. The transformation does not bother about whether they are located in the conceptual model or the robustness model, as long as they share the same type.

The second state contains a call to the *map-externaltypes-controlled-tgg* method that has been used before in the context of Figure 10.7. Figure 10.11 displays the signature of this method. Moreover, it indicates that the method's behavior can be modeled by a single Story Pattern. Again, this rather obvious diagram may be generated by a higher order transformation.

As Figure 10.12 indicates, the Story Pattern from the *map-externaltypes-controlled-tgg* method applies the TGG syntax and therefore has a semantics that is much richer than the patterns presented in Chapter 4. The main difference with Figure 10.6 is that the *ca* and *ra*

Figure 10.13: Reflective TGG rule, assumed to be executed in a loop-state iterating over *ma*.

that without a proper traceability tool, it is hard to understand the complete semantics of a set of TGG rules. It should be noted that the emerging QVT standard [197] proposes a two-level transformation language architecture as well which makes [259] applicable in that context too.

10.3.2 Reflection

Next to the flexibility problem discussed in section 10.2, the case study challenged the TGG formalism as follows: the complex class hierarchy of the UML metamodel resulted in too weak or too verbose rewrite rules. From the investigated techniques, we believe that the use of runtime reflection has the most potential for being useful in other applications of triple graph grammars.

10.3.2.1 Reflective Triple Graph Grammars

Figure 10.13 illustrates how the *<< dynamically-typed >>* stereotype allows transformation writers to abstract from the concrete type of a model element: *classInCM* and *entityInRM* could be instances of *UmlClass* or *Datatype*. The advantage of using the reflective *refGetValue* calls in the state constraint (shown at the bottom) of the figure is not only that transformation writers are not bothered with explicitly enumerating all properties (*name*, *visibility*, ...) of these metaclasses: without this use of runtime type information, one would need to write one TGG rule for *classInCM* and *entityInRM* being instances of *UmlClass* and a very similar yet slightly different one for the *Datatype* case.

Note that technically, the *<< reflective >>* class links from *classInCM* and *entityInRM* to *mc* may need a different treatment than other links since not all modeling platforms may support the dynamic modification of a model element's type. More specifically, while a *CLOS* metaprogramming environment supports intercession [28], a *Java/JMI* solution like *NetBeans MDR* does not support this feature [169]. Such details are hidden to transformation writers.

Figure 10.14: Non-reflective version of the TGG rule shown on Figure 10.13. Usage of a dedicated operator for equivalence across all properties of a model element.

Finally, note the use of the *ignoreConstraints* property in the state constraint. Again, its effect is that modelers are given the freedom to ignore the consistency constraint of this TGG rule in a controlled manner. The *cp* and *rp* nodes are bound using a two-state transformation similar to the one presented in Figure 10.9.

10.3.2.2 Reflection Alternative

Instead of modeling the lookup of *classInCM*'s and *entityInRM*'s *ma* properties and the potential modification of the *class* link to *mc* explicitly, a transformation writer may want to hide these mechanisms by using a language that has built-in operators for stating the desired properties about *classInCM* and *entityInRM*.

Figure 10.14 presents how the TGG rule discussed in the previous paragraph could look when using an operator called *<<equal-by-value>>*. This operator expresses that the nodes connected by dotted arrows should be an instance of the same type and should have the same attribute values.

Chapter 7 discussed a similar operator in more detail. Chapter 8 proposed static (“*compiler*”) mechanisms for translating the complex operator back into more primitive language constructs. The example given above illustrates that a dynamic (“*interpreter*”) approach is needed when the concrete type of the nodes is not known at rule deployment time. More specifically, the *<<create>>* stereotypes in Figure 10.14 cannot be statically transformed into conventional (i.e., non-TGG) *<<create>>* stereotypes since that would imply that direct instances of *Classifier* would be created. That would not only be undesirable (due to the loss of properties defined in *UmlClass* or *Datatype*) but also impossible (since *Classifier* is an abstract metaclass). A static rule translation approach would of course still be possible if counterparts of the *<<dynamically-typed>>* and *<<reflective>>* operators (see part 10.3.2.1) would be available in the target transformation language.

10.4 Related Work

The proposed kind of transformation models combine characteristics from two programming paradigms. This section first relates the features from the *hybrid* transformation modeling style to other work on rule-based graph and controlled graph rewriting. Then, it points to other work on hybrid transformation languages.

Rule-Based Graph Rewriting The proposed modeling style is based on conventional Triple Graph Grammars, a strictly rule-based formalism. Even after the inclusion of control-flow constructs, the formalism can still be used in a rule-based manner: the rule from Figure 10.7, for example, takes care of only two inconsistency types. As discussed in part 10.3.1.1, other rules will be triggered in the other cases. It should be noted that the order between rules can be of importance.

In the example given, the rule from Figure 10.7 should be executed before those of Figures 10.5 and 10.6. In contrast to the standard TGG formalism [229], this order cannot be derived from an inspection of the rules.

Becker et al. proposed an extension of the Triple Graph Grammar semantics to accommodate developer interaction [18]: in a first phase, all rules are tested for a match. If multiple matches can be found, developers are required to choose which rewrite rule should be executed in the subsequent phase. In that approach, there is no fundamental need for embedding the TGG rules into a control flow.

The approach presented in this chapter is more flexible in that (1) decision processes spanning multiple interactions with developers can be modeled easily using multiple user interaction calls and (2) the decision of the developer can be supported by highlighting explicitly defined sets of elements. In Becker's approach, only one model element (per rule), called the *dominant increment*, can be highlighted to developers.

Controlled Graph Rewriting Control flow has been added on top of the TGG formalism by allowing the use of $\ll bound \gg$ nodes in TGG rules. These $\ll bound \gg$ nodes can be introduced by all conventional means defined by Story Diagram language: (1) sequential composition, (2) operation parameter passing and (3) use of the *this* instance. Other parts of the case study that are not explicitly documented in this chapter indicate that other features of Story Diagrams, like *negative application conditions* [282], are useful in the context of hybrid rules as well.

As stated in the introduction, the emerging QVT standard refers to the combination of its declarative and operational sub-languages as a hybrid transformation language. However, the combination of these sub-languages has not been demonstrated on an actual transformation problem yet. The most concrete proposal of integrating declarative language features with operational ones to date may have been published by Jouault and Kurtev [127].

Their *ATLAS Transformation Language* (ATL) allows one to map elements in *called rules* with the same syntax as that for *matched rules*. Matched rules can be compared with graph grammar rules. Called rules make no use at all of a matching engine. Therefore, ATL could not be employed as the hybrid transformation language introduced in this chapter: in part 10.3.1.1 we analyzed under what condition two declarative rules would result in an ambiguity and resolved it by defining an operational rule with higher precedence. This operational rule used input from the modeler (which would also be supported by ATL's *native* called rules) to delegate to the proper declarative rule (which would also be supported by ATL).

ATL's limitation is that the operational rule cannot be scheduled between (or in this case: *before*) declarative rules. It should also be noted that the main declarative power of TGG rules is that they support bidirectional and incremental consistency maintenance with very low specification effort. To overcome its limitations, declarative ATL may follow an evolution comparable to that from pair graph grammars to triple graph grammars, as discussed in Subsection 10.2.2.1.

10.5 Summary and Outlook

This chapter aimed to close this third part of the thesis with pointers to promising future work on transformation modeling. The complexity of the transformation problem under study consisted of the fact that one inconsistency could be resolved in many ways. In particular, user-friendly developer interaction was needed to resolve ambiguities in the conflict resolution process. After illustrating that neither a purely operational, nor a purely declarative approach was sufficient to model the transformation in a desirable manner, the integration of the two approaches was investigated. The proposed modeling style is hybrid in the sense that within transformation rules, constructs such as branches and method calls are supported whereas the scheduling between rules remains implicit by default. Moreover, change propagation and the execution direction remain implicit too.

In our ongoing work, we are comparing the hybrid transformation language approach presented in this chapter with an approach that maps fully declarative transformation models on operational transformation models [259]. The ambiguities discussed in this chapter can then be resolved at the level of the operational transformation models.

Summary and Conclusions

Chapter 11

Summary and Conclusions

This chapter briefly presents the general conclusions from this thesis. Although all observations are supported by the chapters that contain the actual contributions, this concluding chapter does not aim to summarize individual chapters. Instead, such summaries can be found at the end of each preceding chapter. The conclusions are presented in three steps: Section 11.1 revisits the research background and Section 11.2 puts the solutions from the previous chapters in a broader context while Section 11.3 finally concludes this text.

11.1 Original Motivation

The research that has lead to this thesis emerged from my personal dissatisfaction about existing approaches to the platform independent development of enterprise applications in 2002. More specifically, in my master's thesis, I had investigated the market of J2EE application servers both horizontally (by creating a feature matrix for more than 20 products) and vertically (by performing a detailed comparison of two products). When graduating, I could not accept the amount of effort required to deploy a rather simple application on these products that were supposed to represent the state-of-the-art in platform independent application programming. In fact, it seemed that the application server vendors deliberately did not support full framework interoperability to maintain a competitive advantage. More specifically, apart from the standard J2EE artifacts, realistic enterprise applications appeared to require several vendor-specific artifacts [261].

Therefore, I accepted the hypothesis that in order to achieve more platform independence, a generative approach was needed. Interestingly, the MDA initiative promised to support this approach with standards: by enabling developers to extend standard modeling languages as well as the mappings to execution platforms, one would be able to overcome technical differences at the implementation level more easily.

Unfortunately, it turned out that essential standards were missing to realize this vision in 2002. Still, the MDA initiative remained inspiring due to the early industrial adoption of the underlying model-driven engineering paradigm. While most early success stories were based on the application of model-to-text transformation technology for structural models [190], meanwhile more challenging transformations of behavioral models have been applied in practice too [130].

However, until 2005 there was no standard way to define model transformations and the preliminary QVT standard has only been applied in isolated pilot projects [118]. Therefore, most “MDA” success stories are based on proprietary solutions at the transformation level, which obviously beats the purpose of being platform independent. In fact, while I was realizing that even the MDA initiative implied the risk of vendor lock-in at the transformation level, Miguel Angel de Miguel Cabello reported that problem at the seminal Workshop in Software Model Engineering at the UML 2002 conference [55].

11.2 Solutions

In this thesis, I propose several solutions to that specific problem. It should be noted that I have not aimed to “compete” with industrial MDA projects that continued to apply the existing modeling and transformation techniques on a large scale. Instead, the fundamentals of different transformation approaches were considered in detail. The result of this consideration is reflected in Chapter 2 in the form of a taxonomy of model transformations.

After considering the, often premature, proprietary transformation solutions, I investigated whether the UML would be sufficient to model transformations. After all, this language consists of various diagram types that had been studied and applied for years. Two aspects of the UML were considered in detail: one the one hand, the Object Constraint Language has been applied in the context of refactoring and model synchronization. The OCL is a textual language supporting the object-oriented design by contract paradigm. On the other hand, a special combination of class and activity diagrams is proposed to complement the limitations of that modeling style.

This solution not only solves the lock-in problem discussed above: several case study evaluations confirm that the modeling style has interesting advantages from a cognitive perspective too. Moreover, the supportive formalism has even been extended with cognitive concerns in mind. While Section 1.1.3 already summarized the core contributions, they are briefly related to concrete chapters here.

Although standard OCL is useful for modeling transformation contracts (cfr. Chapters 3 and 9), the transformation behavior often needs to be modeled more constructively. Chapter 4 illustrates how the latter kind of behavioral models can be created with Story Diagrams. This existing language provided a promising basis for this thesis since it combines mature graph transformation semantics with a syntax that resembles standard UML diagrams.

Chapter 5 discusses how this approach has been aligned with the official UML standard. Moreover, the chapter discusses how an industrial model-to-text transformation tool has been used for automating the transformation from such Story Diagrams into executable code. In turn, the generated code complies with an MDA standard. Chapter 6 investigates transformation model portability in more detail. More specifically, the chapter illustrates how refinements can be modeled such that platform issues are separated from the core transformation behavior.

What may not have been pointed out enough before, is that the proposed techniques for transformation modeling also apply to the platform independent development of end-user applications:

Use of a standard modeling language. The use of a standard modeling language ensures that transformation models can be exchanged between modeling tools. This principle not only applies to transformation models. In fact, the MDA guide proposes the UML as a promising basis for modeling end-user applications [175].

Customization of that modeling language. In order to minimize the mapping between language constructs and mental representations, the standard modeling language often needs to be customized. Chapter 5 introduced a Story Diagram specific profile for this purpose. Similarly, enterprise applications tend to rely on the profile for EDOC [191] or ODP [219].

Extension of that modeling language. Chapter 7 highlights a weakness of the existing languages for controlled graph transformation and illustrates how that weakness was overcome without falling into transformation modeling tool specificities. Chapter 8 elaborates that language extension experiment by discussing the novel way in which the extended language was made executable: a higher order transformation maps the new transformation language constructs back to existing ones. In the context of end-user application modeling, such a “desugaring” approach has been illustrated by Muliawan [177].

Platform mapping. The primary purpose of software modeling remains the construction and maintenance of correctly running systems. Therefore, transformations that map platform independent models to analysis or execution platforms are needed. The *UML-to-CSP* case study from Chapter 6 and the *CM2RM* case study from Chapter 7 are examples of such transformations for end-user applications. The templates from Chapter 5 and the *Copy2GT* case study from Chapter 8 realize platform mappings at the transformation level.

Robustness. The solution to the *UML-to-CSP* case study from Chapter 6 illustrates how the robustness of platform mappings can be supported. The *UML-to-CSP* input models are business process models of end-user applications. However, the proposed technique also applies to transformation models: since transformation models need to be transformed too (cfr., Chapter 8), they may first need to be normalized into a standard form too.

Views. Several chapters illustrated how views can decompose the complexity of large rewrite rules. Modeling tools maintain the consistency between such transformation views automatically. In fact, it is the use of a mature modeling language such as the UML that enables one to leverage features that have been constructed for end-user application modeling in a transformation context too.

Chapters 9 and 10 investigated to what extent the proposed techniques support the complex type of “synchronization” transformations. It turns out that OCL as well as Story Diagrams can contribute to the standardization of that domain too. The latter two chapters are not yet fully supported by the toolset that supports the core contributions of the thesis. However, these chapters motivate that the construction of such a toolset is worthwhile.

11.3 Final Conclusion

In summary, this thesis argues that one should not treat transformations as software artifacts that require completely new languages and/or development techniques. Instead, the development of transformations should be supported more by techniques that are successful in the context of end-user application development.

More specifically, this thesis illustrates how techniques that emerged in the context of model-driven engineering for end-user applications can support the development of the supportive model transformations too. In particular, the thesis shows how transformations can be modeled and transformed with standard versions of existing languages. This reduces the total amount of technologies that industrial software engineers need to master. Moreover, research on application modeling in general can now be aligned further with research on model transformations. This reduces the chance that separate communities work on the same fundamental problems without realizing this.

At the end of each chapter, several issues have been identified as open research challenges. In the context of refactoring for example, it still needs to be investigated how one can automatically compose refactoring operations based on standard OCL contracts. There are several active projects in the area of standards-compliant model transformation such that progress is being made each day [4, 223, 80, 36, 96]. Personally, I will investigate in more depth how complex graph operations such as copying can be expressed more naturally. Moreover, I will work on hybrid transformation modeling languages that support implicit as well as explicit rule scheduling. Finally, I want to stimulate collaboration within and across the model transformation community by maintaining the model transformation taxonomy.

Bibliography

- [1] Software Acumen. MDA Generators. <http://www.codegeneration.net/generators-by-standard.php?standard=1>, 2007. (p. 43)
- [2] David Akehurst, Stuart Kent, and Octavian Patrascoiu. A Relational Approach to Defining and Implementing Transformations in Metamodels. *Software and Systems Modeling*, 2(4):215–239, December 2003. (p. 28, 29, 30, 31, 181)
- [3] David H. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, University of Kent at Canterbury, December 2000. (p. 27, 28, 181)
- [4] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In Rensink and Warmer [211], pages 361–375. (p. 171, 206)
- [5] C. Amelunxen and A. Schürr. Formalizing Model Transformation Rules for UML/-MOF 2. *IET Software Journal*, 2007. accepted for publication. (p. 22)
- [6] Paul Anka and Frank Sinatra. *My way*, 1968. (p. 3)
- [7] Robert S. Arnold. *An introduction to software restructuring*, pages 228–269. IEEE Press, 1986. (p. 68)
- [8] Márk Asztalos, László Lengyel, Tihámér Levendovszky, and Hassan Charaf. Termination analysis of the transformation UML to CSP. In *8th International Symposium of Hungarian Researchers on Computational Intelligence and Informatics*, 2007. (p. 49)
- [9] Thomas Baar, editor. *OCLE Workshop - Tool Support for OCL and Related Formalisms - Needs and Trends, MoDELS'05 Conference Workshop*, Technical Report LGL-REPORT-2005-001. EPFL, October 2005. (p. 14)
- [10] Thomas Baar. OCL and graph-transformations – a symbiotic alliance to alleviate the frame problem. In Bruel [35], pages 20–31. (p. 56)
- [11] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer Verlag, 1998. (p. 43)
- [12] Jason Baker and Wilson C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281, New York, USA, 2002. ACM. (p. 59)

- [13] Robert Balzer. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. (p. 179, 181)
- [14] Aline Lúcia Baroni. Formal definition of object-oriented design metrics. Master's thesis, Vrije Universiteit Brussel - Belgium and Faculty of Sciences in Collaboration with Ecole des Mines de Nantes - France and Universidade Nova de Lisboa - Portugal, 2002. (p. 75)
- [15] Meyers Bart, Van Den Enden David, and Lamkanfi Ahmed. Verslag transformaties met MoTMoT. In *Modeling and Transformation in Software Development* [124]. (p. 154)
- [16] Victor R. Basili and Barry Boehm. COTS-based systems top 10 list. *Computer*, 34(5):91–93, 2001. (p. 97)
- [17] S. Becker, T. Haase, and B. Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *Journal of Software and Systems Modeling*, 4:123–140, 2004. (p. 176, 186)
- [18] Simon M. Becker, Sebastian Lohmann, and Bernhard Westfechtel. Rule execution in graph-based incremental interactive integration tools. In *Proc. Intl. Conf. on Graph Transformations*, volume 3256 of *LNCS*, pages 22–38. Springer, 2004. (p. 188, 199)
- [19] Catriel Beeri, Alberto O. Mendelzon, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalence of relational database schemes. In *STOC'79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 319–329, New York, NY, USA, 1979. ACM. (p. 43)
- [20] Berthold Hoffmann. Abstraction and Control for Shapely Nested Graph Transformation. *Fundamenta Informaticae*, 58(1):39–65, 2003. (p. 145)
- [21] Bertrand Meyer. *Eiffel, the Language*. Prentice Hall, 1992. (p. 174)
- [22] Jean Bézivin. Model Driven Engineering: An emerging technical space. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 2006. (p. 51)
- [23] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, October 2003. (p. 50, 173)
- [24] Jean Bézivin and Reiko Heckel. 04101 summary – language engineering for model-driven software development. In Bezivin and Heckel [25]. (p. 42)
- [25] Jean Bezivin and Reiko Heckel, editors. *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. (p. 208, 219)

- [26] Dénes Bisztray, Karsten Ehrig, and Reiko Heckel. Case study: UML to CSP transformation. Technical report, University of Leicester, UK, 2007. (p. 108, 111, 113, 115, 117, 131)
- [27] Alan F. Blackwell. Ten years of cognitive dimensions in visual languages and computing: Guest editor's introduction to special issue. *Journal of Visual Languages and Computing*, 17:285–287, August 2006. (p. 57, 84)
- [28] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. *Object-oriented programming: the CLOS perspective*, chapter CLOS in context: the shape of the design space, pages 29–61. MIT Press, Cambridge, MA, USA, 1993. (p. 197)
- [29] M. Bohlen. AndromDA - from UML to Deployable Components, version 2.1.2, 2003. <http://andromda.sourceforge.net>. (p. 94)
- [30] Artur Boronat and Reiko Heckel. UML models to CSP models in MOMENT2-GT. In *AGTIVE 2007 Tool Contest Solutions* [213]. (p. 49)
- [31] Youcef Bouchebaba, Bruno Girodias, Gabriela Nicolescu, El Mostapha Aboulhamid, Bruno Lavigueur, and Pierre Paulin. Mpsoc memory optimization using program transformation. *ACM Trans. Des. Autom. Electron. Syst.*, 12(4):43, 2007. (p. 43)
- [32] David Braun, Jeff Sivils, Alex Shapiro, and Jerry Versteegh. History of UML. http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/history_of_uml.htm, 2001. (p. 14)
- [33] Jr. Frederic P. Brooks. *The Mythical Man Month*. Addison–Wesley, anniversary edition, 1995. ISBN: 0–201–83595–9. (p. 37)
- [34] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. Semantic issues of OCL: Past, present, and future. *Electronic Communications of the EASST*, 5, 2006. (p. 16)
- [35] Jean-Michel Bruel, editor. *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*. Springer, 2006. (p. 207, 216, 220)
- [36] Thomas Buchmann, Alexander Dotor, and Bernhard Westfechtel. Model-driven development of graphical tools. In *ICSOFIT*, Barcelona, Spain, 2007. (p. 206)
- [37] Fabian Büttner and Martin Gogolla. Realizing UML metamodel transformations with AGG. In Reiko Heckel, editor, *Proc. ETAPS Workshop Graph Transformation and Visual Modeling Techniques (GT-VMT'2004)*, ENTCS. Elsevier, 2004. (p. 147)
- [38] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. Demonstration at the 22th Int. Conf. on Automated Software Engineering (ASE'07), November 2007. (p. 75)
- [39] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. Model transformation contracts and their definition in UML and OCL. Technical Report 2004-08, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, 59655 Villeneuve d'Ascq Cedex, France, 2004. (p. 25)

- [40] Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the specification of model transformation contracts. In Patrascoiu [203], pages 69–83. (p. 25, 26)
- [41] Jarinee Chattratchart and Jasna Kuljis. Some evidence for graphical readership, paradigm preference, and the match-mismatch conjecture in graphical programs. In *PPIG – 13th Annual Workshop of the Psychology of Programming Interest Group*, pages 173–189, Bournemouth University, UK, apr 2001. (p. 57)
- [42] D. Chiorean, A. Carcu, M. Pasca, C. Botiza, H. Chiorean, and S. Moldovan. *Studia Informatica*, volume XLVII, chapter UML Model Checking, pages 71–88. Babes-Bolyai University, 400084 Cluj-Napoca, Romania, 2002. (p. 143)
- [43] D. Chiorean et al. Object constraint language environment (OCLE), version 2.02, 2004. <http://lci.cs.ubbcluj.ro/ocle>. (p. 101)
- [44] Dan Chiorean, Birgit Demuth, Martin Gogolla, and Jos Warmer. OCL workshop - OCL for (meta-)models in multiple application domains (OCLApps 2006). In Tiziana Margaria, Julia Padberg, and Gabriele Tüntzer, editors, *Electronic Communications of the EASST*, volume 5, 2006. (p. 14)
- [45] Tony Clark and Jos Warmer. OCL Workshop - UML 2.0 - the future of the UML Object Constraint Language (OCL). In John Knapman, Alfred Strohmeier Bran Selic, and David Garlan, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference*, volume 1939 of *Lecture Notes in Computer Science*. Springer, October 2000. (p. 14)
- [46] NetBeans Community and Inc. Sun Microsystems. Java-model...mdr...metamodel...refobjects...multipartids...what is all this stuff? <http://wiki.netbeans.info/wiki/view/DevFAQJavaModel>, 2007. (p. 153)
- [47] James R. Cordy, Ralf Lämmel, and Andreas Winter. 05161 executive summary – transformation techniques in software engineering. In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. (p. 42)
- [48] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, and Anika Wagner. *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*, pages 163–245. Volume I: Foundations of Rozenberg [221], 1997. (p. 34)
- [49] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006. (p. 42, 44, 48, 55, 58)
- [50] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003. (p. 15)

- [51] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. *splc*, 0:41–51, 2006. (p. 55)
- [52] David Crocker. Safe object-oriented software: the verified design-by-contract paradigm. In F.Redmill and T.Anderson, editors, *Proceedings of the Twelfth Safety-Critical Systems Symposium*, pages 19–41. Springer-Verlag, 2004. (p. 174)
- [53] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Täntzer. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.*, 376(3):139–163, 2007. (p. 16)
- [54] Miguel A. de Miguel, Daniel Exertier, and Serge Salicki. Specification of model transformations based on meta templates. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering*, 2002. (p. 15)
- [55] M. deMiguel, D. Exertier, and S. Salicki. Specification of model transformations based on meta templates. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering (WiSME) – Satellite event of the Fifth International Conference on the Unified Modeling Language*, 2002. (p. 204)
- [56] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005. (p. 13)
- [57] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM. (p. 167)
- [58] D.H.Akehurst and A.G.Waters. UML Deficiencies from the perspective of automatic Performance Model Generation. In *OOPSLA’99 Workshop on Rigorous Modelling and Analysis with the UML: Challenges and Limitations*, November 1999. (p. 16)
- [59] Marcos Didonet Del Fabro and Frédéric Jouault. Model transformation and weaving in the AMMA platform. In *Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 71–77. Springer, July 2005. (p. 43)
- [60] Ira Diethelm, Leif Geiger, Thomas Maier, and Albart Zundorf. Turning collaboration diagram strips into storycharts. In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE’02*, Orlando, Florida, USA, 2002. (p. 34)
- [61] Ira Diethelm, Leif Geiger, and Albert Zündorf. Systematic story driven modeling, a case study (library system). Technical report, University of Kassel, 2004. (p. 34)
- [62] Ira Diethelm, Leif Geiger, and Albert Zündorf. Systematic story driven modeling, a case study (Paderborn shuttle system). In *Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE’04*, Edinburgh, Scotland, 2004. (p. 34, 132)
- [63] Ira Diethelm, Leif Geiger, and Albert Zündorf. Teaching modeling with objects first. In *WCCE 2005, 8th World Conference on Computers in Education*, Cape Town, South Africa, 2005. (p. 33)

- [64] Remco M. Dijkman. YATL4MDR: A model transformation engine and OCL checker for the NetBeans meta-data repository. <http://is.tm.tue.nl/staff/rdijkman/yatl4mdr.html>. (p. 101, 171, 180)
- [65] Brian Dobing and Jeffrey Parsons. How UML is used. *Commun. ACM*, 49(5):109–113, 2006. (p. 16)
- [66] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64:249–283, 2002. (p. 145)
- [67] Christopher Dutchyn, Paul Lu, Duane Szafron, Steven Bromling, and Wade Holst. Multi-dispatch in the Java virtual machine: Design and implementation. In Raj and Wang [208], pages 77–92. (p. 59)
- [68] Niels Van Eetvelde. *A Graph Transformation Approach to Refactoring*. PhD thesis, University of Antwerp, June 2007. (p. 153)
- [69] Niels Van Eetvelde and Dirk Janssens. Extending graph rewriting for refactoring. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *2nd Int'l Conference on Graph Transformation*, volume 3256 of *Lecture Notes in Computer Science*, pages 399–415. Springer-Verlag, 2004. (p. 153)
- [70] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of graph grammars and computing by graph transformation*, volume II: Applications, Languages, and Tools. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999. (p. 213, 224)
- [71] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic approaches to graph transformation. Part II: single pushout approach and comparison with double pushout approach*, pages 247–312. Volume I: Foundations of Rozenberg [221], 1997. (p. 34)
- [72] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, pages 1–69, London, UK, 1979. Springer-Verlag. (p. 16)
- [73] Hartmut Ehrig and Karsten Ehrig. Overview of formal concepts for model transformations based on typed attributed graph transformation. In *Proc. Int'l Workshop on Graph and Model Transformation (GraMoT 2005)*, Electronic Notes in Computer Science. Elsevier, 2005. (p. 14)
- [74] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Tántzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005. (p. 56)
- [75] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundam. Inf.*, 74(1):135–166, 2006. (p. 16)

- [76] Hartmut Ehrig, Julia Padberg, Ulrike Prange, and Annegret Habel. Adhesive high-level replacement systems: A new categorical framework for graph transformation. *Fundam. Inf.*, 74(1):1–29, 2006. (p. 16)
- [77] Klaus-D. Engel, Richard F. Paige, and Dimitrios S. Kolovos. Using a model merging language for reconciling model versions. In Rensink and Warmer [211], pages 143–157. (p. 43)
- [78] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments. part I: tool specification. *ACM Trans. Softw. Eng. Methodol.*, 1(2):135–167, 1992. (p. 188)
- [79] C. Ermel, M. Rudolf, and G. Taentzer. *The AGG approach: language and environment*, pages 551–603. Volume II: Applications, Languages, and Tools of Ehrig et al. [70], 1999. (p. 55)
- [80] Claudia Ermel and Karsten Ehrig. Visualization, simulation and analysis of reconfigurable systems. In *Workshop on Application of Graph Transformation with Industrial Relevance (AGTiVE)*, October 2007. (p. 206)
- [81] Jean-Marie Favre et al. Planet MDE, Model Driven Engineering. <http://www.planetmde.org/>, 2007. (p. 13, 14)
- [82] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In Martin Gogolla, Paul Sammut, and Jon Whittle, editors, *Proc. 3rd Workshop in Software Model Engineering (Satellite workshop at the 7th International Conference on the UML)*, October 2004. (p. 14)
- [83] M.S. Feather, S. Fickas, A. Finkelstein, and A. van Lamsweerde. Requirements and specification exemplars. *Automated Software Engineering*, 4(4):419–438, 1997. (p. 17)
- [84] Anthony Finkelstein. A foolish consistency: Technical challenges in consistency management. In *Proceedings of the 11th International Workshop on Database and Expert Systems Applications*, 2000. (p. 191)
- [85] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multi-perspective specifications. In *ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 84–99, London, UK, 1993. Springer-Verlag. (p. 181)
- [86] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph RewriteLanguage Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, Nov 1998. (p. 89)
- [87] R. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symposium on Mathematical Aspects of Computer Science*, volume 19, pages 19–32. American Mathematical Society, 1967. (p. 174)
- [88] Brian Foote, Ralph E. Johnson, and James Noble. Efficient multimethods in a single dispatch language. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 337–361. Springer, 2005. (p. 59)

- [89] National Committee for Information Technology Standards. Programming language Smalltalk. Document Number: ANSI/INCITS 319-1998, 1998. (p. 49)
- [90] Timo Du Four and Ruben De Smet. extending eDOBS. <https://lists.uni-paderborn.de/pipermail/fujaba-developers/2005q4/000393.html>, 2005. (p. 80)
- [91] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, July 1999. (p. 43, 68, 69, 71, 75, 76)
- [92] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999. (p. 80, 154)
- [93] FOLDOC Free Online Dictionary Of Computing. Declarative language. <http://foldoc.org/index.cgi?query=declarative+language>, 2007. (p. 52)
- [94] Fujaba Developers. Fujaba Associations Specification. <http://wwwcs.uni-paderborn.de/cs/fujaba/documents/developer/guidelines/assocs/index.html>, March 2007. (p. 89, 94)
- [95] Leif Geiger and Albert Zündorf. Transforming graph based scenarios into graph transformation based junit tests. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2003. (p. 33)
- [96] Tom Gelhausen, Derre Bugra, and Rubino Geiř. Customizing grgen.net for model transformation. In *Workshop on Graph and Model Transformation (GraMoT)*, 2008. accepted for publication. (p. 206)
- [97] Gentleware. Poseidon for UML, version 2.2. <http://www.gentleware.com>, 2004. (p. 95, 99)
- [98] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer-Verlag, 2002. Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain. (p. 41)
- [99] Carlo Ghezzi and Gian Pietro Picco. An outlook on software engineering for modern distributed systems. In *Proceedings of the Monterey workshop on Radical Approaches to Software Engineering*, Venice (Italy), August 2002. (p. 13)
- [100] Holger Giese, Albert Zündorf, Andy Schürr, Bernhard Westfechtel, and Leif Geiger. Fujaba toolsuite. In *International Fujaba Days*, 2003–2007. (p. 33, 34, 79)
- [101] R. L. Glass. Maintenance: Less is not more. *IEEE Software*, July 1998. (p. 68)
- [102] Martin Gogolla and Mark Richters. Transformation rules for UML class diagrams. In *UML'98: Selected papers from the First International Workshop on The Unified Modeling Language*, pages 92–106, London, UK, 1999. Springer-Verlag. (p. 147)

- [103] Jean Graef. Managing taxonomies strategically. *Montague Institute Review*, March 2001. (p. 41)
- [104] T. R. G. Green. Cognitive dimensions of notations. In Alistair Sutcliffe and Linda Macaulay, editors, *People and Computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press. (p. 57)
- [105] T. R. G. Green and M. Petre. When Visual Programs are Harder to Read than Textual Programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics)*, 1992. (p. 57)
- [106] T. R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996. (p. 57, 84, 86)
- [107] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM Press. (p. 14, 39)
- [108] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, August 1991. (p. 43, 68)
- [109] T. Guimaraes. Managing application program maintenance expenditure. *Comm. ACM*, 26(10):739–746, 1983. (p. 68)
- [110] Annegret Habel, Reiko Heckel, and Gabriele Tüntzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996. (p. 55)
- [111] M. H. Halstead. NELIAC. *Communications of the ACM*, 6(3):91–92, 1963. (p. 164)
- [112] M. Hanna. Maintenance burden begging for a remedy. *Datamation*, pages 53–63, 1993. (p. 68)
- [113] Hartmunt Ehrig and Michael Pfender and Hans Jürgen Schneider. Graph-Grammars: An Algebraic Approach. In *FOCS*, pages 167–180, 1973. (p. 182)
- [114] Martin Hitz and Gerti Kappel. Developing with UML - some pitfalls and workarounds. In *UML'98: selected papers from the first international workshop on the Unified Modeling Language*, volume 1618 of *Lecture Notes in Computer Science*, pages 9–20, London, UK, 1999. Springer-Verlag. (p. 16)
- [115] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (p. 174)
- [116] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2:335–355., 1973. (p. 174)
- [117] Berthold Hoffmann, Dirk Janssens, and Niels Van Eetvelde. Cloning and expanding graph transformation rules for refactoring. In *International Workshop on Graph and Model Transformation*, Tallinn, Estonia, 2005. A satellite event of GPCE'05. (p. 153)

- [118] Patrick C. K. Hung, editor. *EDOC'06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, Hong Kong, October 2006. IEEE Computer Society. (p. 204, 226)
- [119] Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification i: A logical basis and its implementation. *Acta Informatica*, 4:145–182, 1974. (p. 174)
- [120] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPSLA*, pages 347–349, 1986. (p. 59)
- [121] Software Engineering Institute. Cyclomatic complexity. http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html. (p. 76)
- [122] Jens H. Jahnke, Wilhelm Schäfer, Jörg P. Wadsack, and Albert Zündorf. Supporting iterations in exploratory database reengineering processes. *Sci. Comput. Program.*, 45(2-3):99–136, 2002. (p. 33)
- [123] J. Jakob, A. Königs, and A. Schürr. Non-materialized Model View Specification with Triple Graph Grammars. In A. Corradini, editor, *International Conference on Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 321–335, Natal, Brazil, 2006. Springer Verlag. (p. 52)
- [124] Dirk Janssens and Pieter Van Gorp. Modeling and transformation in software development. <http://www.ua.ac.be/>, 2007. (p. 154, 208, 217, 219)
- [125] Jeffrey D. Ullman and Jennifer Widom. *A first course in database systems*, chapter 3.3, pages 103–107. Prentice-Hall, 1997. (p. 18)
- [126] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1188–1195, New York, NY, USA, 2006. ACM. (p. 50)
- [127] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Bruel [35], pages 128–138. (p. 175, 186, 199)
- [128] Frédéric Jouault, William Piers, Dennis Wagelaar, Milan Milanovich, et al. How can i implement my own injector or extractor? <http://wiki.eclipse.org/>. (p. 51)
- [129] Gabor Karsai and Gabriele Tántzer, editors. *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *Electronic Notes in Theoretical Computer Science*. Elsevier, March 2006. (p. 219)
- [130] O. Kath, A. Blazarenas, M. Born, K.-P. Eckert, M. Funabashi, and C. Hirai. Towards executable models: Transforming EDOC behavior models to CORBA and BPEL. *EDOC*, 00:267–274, 2004. (p. 203)
- [131] Anne Keller and Serge Demeyer. A meta-model approach to inconsistency management. In *International Workshop on Object-Oriented Reengineering (WOOR) – Satellite event of the 21st European Conference on Object-Oriented Programming*, 2007. (p. 43)

- [132] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An Adaptable TGG Interpreter for In-Memory Model Transformation. In Andy Schürr and Albert Zündorf, editors, *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, volume tr-ri-04-253 of *Technical Report*, pages 35–38. University of Paderborn, September 2004. (p. 188)
- [133] Joris Kint, Tristan Lambert, and Erik Vanderstraeten. Extract interface. In *Modeling and Transformation in Software Development* [124]. (p. 154)
- [134] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The epsilon object language (EOL). In *Model Driven Architecture Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006. (p. 173)
- [135] A. Königs and A. Schürr. MDI - a Rule-Based Multi-Document and Tool Integration Approach. *Special Section on Model-based Tool Integration in Journal of Software and System Modeling*, 2005. accepted for publication. (p. 188)
- [136] A. Königs and A. Schürr. Tool integration with Triple Graph Grammars - a survey. In R. Heckel, editor, *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Amsterdam, 2006. Elsevier Science Publ. (p. 188)
- [137] Jernej Kovse and Theo Härder. Generic XMI-based UML model transformations. In *OOIS'02: Proceedings of the 8th International Conference on Object-Oriented. Information Systems*, pages 192–198, London, UK, 2002. Springer-Verlag. (p. 14)
- [138] R. Kramer. iContractor - The Java Design by Contract Tool. In *Technology of Object-Oriented Languages and Systems (26th International Conference and Exhibition)*, 1998. (p. 174)
- [139] Hans-Jörg Kreowskia, Karsten Hölschera, and Peter Knirscha. Semantics of visual models in a rule-based setting. In Reiko Heckel, editor, *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 75–88, Dagstuhl, Germany, March 2004. Elsevier. (p. 155)
- [140] C. Record L. Geiger, C. Schneider. Template- and modelbased code generation for MDA-tools. In *Proc. 3rd International Fujaba Days*, Paderborn, Germany, September 2005. (p. 100)
- [141] A. V. Lamsweerde, R. Darimont, and P. Massonet. The Meeting Scheduler System - Problem Statement. Technical report, Université Catholique de Louvain - Département d'Ingénierie Informatique, B-1348 Louvain-la-Neuve (Belgium), 1992. (p. 17)
- [142] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, chapter Using Graph Transformation for Practical Model Driven Software Engineering. Springer-Verlag, 2005. Edited by Sami Beydeda and Volker Gruhn. (p. 34)

- [143] Olivier Lecarme, Mireille Pellissier, and Marie-Claude Thomas. Computer-aided production of language implementation systems: A review and classification. In *Software: Practice and Experience*, volume 12, pages 785–824, 1982. (p. 164)
- [144] A. Ledeczki, G. Balogh, Z. Molnar, P. Volgyesi, and M. Maroti. Model integrated computing in the large. *Aerospace Conference, 2005*, pages 1–8, 5-12 March 2005. (p. 14, 39, 167)
- [145] Stan Lee. The amazing spider-man. In Stan Lee, editor, *Amazing Fantasy*, volume 1. Marvel Entertainment Group, August 1962. (p. 166)
- [146] Leif Geiger. Design Level Debugging mit Fujaba. In *Informatiktag*, Bad Schussenried, Germany, 2002. der Gesellschaft für Informatik. (p. 60)
- [147] Laszlo Lengyel, Tihamer Levendovszky, Gergely Mezei, Bertalan Forstner, and Hassan Charaf. Metamodel-based model transformation with aspect-oriented constraints. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 111–123, apr 2006. (p. 58)
- [148] B. P. Lientz and E. B. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980. (p. 68)
- [149] Sten Loecher and Stefan Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. *Electronic Notes in Theoretical Computer Science*, 102, November 2004. (p. 101)
- [150] Thomas Maier and Alber Zündorf. Yet another association implementation. In *Second Int. Fujaba Days (FD'04)*, Darmstadt, Germany, 2004. (p. 94)
- [151] Albert Manoli, Javier Muñoz, Vicente Pelechano, and Oscar Pastor. Model to text transformation in practice: Generating code from rich associations specifications. In John F. Roddick, V. Richard Benjamins, Samira Si-Said Cherfi, Roger H. L. Chiang, Christophe Claramunt, Ramez Elmasri, Fabio Grandi, Hyoil Han, Martin Hepp, Miltiadis D. Lytras, Vojislav B. Misic, Geert Poels, Il-Yeol Song, Juan Trujillo, and Christelle Vangenot, editors, *ER (Workshops)*, volume 4231 of *Lecture Notes in Computer Science*, pages 63–72, Tucson, AZ, USA, November 2006. Springer. (p. 43)
- [152] Michele Marchesi and Giancarlo Succi, editors. *Proceedings of the 4th International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2003)*, volume 2675 of *Lecture Notes in Computer Science*. Springer, May 2003. (p. 68)
- [153] Slavisa Markovic. Composition of UML described refactoring rules. In Patrascioiu [203], pages 45–59. (p. 76)
- [154] M. Mazaud, R. Rakotozafy, and A. Szumachowski-Despland. Code generator generation based on template-driven target term rewriting. In *on Rewriting techniques and applications*, pages 105–120, London, UK, 1987. Springer-Verlag. (p. 43)

- [155] Peter McBrien and Alexandra Poulouvassilis. Automatic migration and wrapping of database applications - a schema transformation approach. In *ER '99: Proceedings of the 18th International Conference on Conceptual Modeling*, pages 96–113, London, UK, 1999. Springer-Verlag. (p. 43)
- [156] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures – Special Issue: Smalltalk*, 32(2-3):140–156, 2006. (p. 105)
- [157] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *SEKE'02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 289–296, New York, NY, USA, 2002. ACM. (p. 105)
- [158] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion – a taxonomy of model transformations. In Bezivin and Heckel [25]. (p. 41, 42)
- [159] Tom Mens and Theo D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering Journal*, 7(1):39–59, February 2000. (p. 76)
- [160] Tom Mens, Niels Van Eetvelde, Dirk Janssens, and Serge Demeyer. Formalising refactorings with graph transformations. *Fundamenta Informaticae*, 2003. (p. 69)
- [161] Tom Mens, Carine Lucas, and Patrick Steyaert. Supporting disciplined reuse and evolution of UML models. In J. Bezivin and P.-A. Muller, editors, *Proceedings of « UML » '98 – The Unified Modeling Language*, volume 1618 of *Lecture Notes in Computer Science*, pages 378–392. Springer-Verlag, 1999. Mulhouse, France. (p. 76)
- [162] Tom Mens, Gabriele Tántzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, apr 2005. (p. 182)
- [163] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004. (p. 43)
- [164] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. In Karsai and Tántzer [129], pages 125–142. (p. 41, 42)
- [165] Tom Mens, Pieter Van Gorp, Daniel Varró, and Gabor Karsai. Applying a model transformation taxonomy to graph transformation technology. In Karsai and Tántzer [129], pages 143–159. (p. 41, 42)
- [166] Lode Mertens, Tim Molderez, and Ruud Wiegers. Flatten statechart refactoring - verslag. In *Modeling and Transformation in Software Development* [124]. (p. 154)
- [167] Gergely Mezei, Tihamér Levendovszky, and Hassan Charaf. *Software Engineering Techniques: Design for Quality*, volume 227, chapter An optimizing OCL Compiler for Metamodeling and Model Transformation Environments, pages 61–71. Springer Boston, 2006. (p. 43)
- [168] Sun Microsystems. Java Metadata Interface Specification. <http://jcp.org/en/jsr/detail?id=40>, Jun 2002. document ID JSR-40. (p. 14, 23, 89, 180)

- [169] Sun Microsystems. NetBeans Metadata Repository, 2002. <http://mdr.netbeans.org/>. (p. 94, 132, 197)
- [170] Sun Microsystems. Java ee apis and docs. <http://java.sun.com/javaee/reference/>, Jul 2007. (p. 13)
- [171] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Redbooks. International Business Machines, January 2004. (p. 100)
- [172] Ana Moreira. Workshop 7: Consistency problems in UML-based software development II – workshops at the UML 2003 conference. In Stevens et al. [241], pages 403–409. (p. 14)
- [173] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling. <http://moncs.cs.mcgill.ca/people/mosterman/campam/>, 2007. (p. 14, 39)
- [174] Ben Moszkowski. *Executing temporal logic programs*. Cambridge University Press, New York, NY, USA, 1986. (p. 181)
- [175] Jishnu Mukerji, Joaquin Miller, and Object Management Group. MDA guide v1.0.1. <http://www.omg.org/cgi-bin/doc?omg/03-06-01>, 2006. (p. 204)
- [176] Olaf Muliawan. Reengineering JCMTG to MoTMoT: a migration from androMDA 2 to 3. Master’s thesis, University of Antwerp, Belgium, 2005. (p. 60)
- [177] Olaf Muliawan, Serge Demeyer, and Dirk Janssens. A transformation language for UML-OCL models. Technical report, University of Antwerp, Belgium, December 2006. (p. 205)
- [178] Olaf Muliawan, Hans Schippers, and Pieter Van Gorp. Model driven, Template based, Model Transformer (MoTMoT). <http://motmot.sourceforge.net/>, 2007. (p. 113, 144, 166, 171, 180)
- [179] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On Executable Meta-Languages applied to Model Transformations. In Bruel [35], pages 120–127. (p. 181)
- [180] Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer, 1996. (p. 220, 224)
- [181] Manfred Nagl and Andy Schürr. Summary and specification lessons learned. In Nagl [180], pages 370–377. (p. 33)
- [182] Duncan Neary and Martin Woodward. An experiment to compare the comprehensibility of textual and visual forms of algebraic specifications. *Journal of Visual Languages and Computing*, 13(2):149–175., apr 2002. (p. 57)

- [183] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002. (p. 93)
- [184] J. Niere and A. Zündorf. Using Fujaba for the development of production control systems. In *Proc. of International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE), Kerkrade, The Netherlands*, LNCS 1779. Springer Verlag, 1999. (p. 93)
- [185] No Magic. Magicdraw. <http://www.magicdraw.com/>, 2004. (p. 95)
- [186] No Magic, Inc. MagicDraw technology - XMI 2.1 reader for Netbeans MDR. http://magicdraw.com/main.php?ts=navig&cmd_show=1&menu=technology, 2007. (p. 109)
- [187] Alex Noort. *Multiple-View Feature Modelling with Model Adjustment*. PhD thesis, Delft University of Technology, 2002. (p. 43)
- [188] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance and Evolution: Research and Practice*, 2:157–174, 1990. (p. 68)
- [189] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *Computer*, 33(4):pp. 24–29, 2000. (p. 191)
- [190] Object Management Group. Success stories. http://www.omg.org/mda/products_success.htm. (p. 166, 203)
- [191] Object Management Group. UML profile for EDOC final adopted specification. <http://www.omg.org/cgi-bin/doc?ptc/02-02-05>. (p. 205)
- [192] Object Management Group. Model Driven Architecture (MDA), document number ormsc/2001-07-01. <http://www.omg.org/cgi-bin/doc?ormsc/01-07-01>, July 2001. (p. 14, 39, 186)
- [193] Object Management Group. Meta Object Facility (MOF) specification, version 1.4. <http://www.omg.org/cgi-bin/doc?formal/2002-04-03>, April 2002. (p. 14, 22, 26, 39, 46, 139)
- [194] Object Management Group. MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10. <http://www.omg.org/cgi-bin/doc?ad/02-04-10>, October 2002. (p. 15, 22, 39, 52)
- [195] Object Management Group. Unified Modeling Language (UML), version 1.5. <http://www.omg.org/cgi-bin/doc?formal/03-03-01>, March 2003. (p. 139, 141)
- [196] Object Management Group. XML Metadata Interchange (XMI), v2.0. <http://www.omg.org/cgi-bin/doc?formal/03-05-02>, 2003. (p. 22, 180)
- [197] Object Management Group. MOF 2.0 QVT final adopted specifications. <http://www.omg.org/cgi-bin/doc?ptc/05-11-01>, November 2005. (p. 15, 39, 44, 53, 58, 173, 197)

- [198] Object Management Group. OCL 2.0 Specification. <http://www.omg.org/cgi-bin/doc?ptc/2005-06-06>, June 2005. (p. 14)
- [199] Object Management Group. XML metadata interchange (XMI), v2.1. <http://www.omg.org/cgi-bin/doc?formal/05-09-01>, September 2005. (p. 14)
- [200] University of Paderborn. Fujaba Tool Suite. <http://www.fujaba.de/>, 2007. (p. 33, 79)
- [201] Olaf Multiawan, Pieter Van Gorp, Anne Keller and Dirk Janssens. Executing a standard compliant transformation model on a non-standard platform. In Frédéric Boulanger, Christophe Gaston, and Pierre-Yves Schobbens, editors, *Workshop on Modeling, Validation and Heterogeneity (MoVaH). Workshop at the First IEEE International Conference on Software Testing, verification and validation (ICST)*, Lillehammer, Norway, April 2008. IEEE. accepted for publication. (p. 107)
- [202] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. (p. 68, 74, 76)
- [203] Octavian Patrascoiu, editor. *OCL and Model Driven Engineering, UML 2004 Conference Workshop*, Lisbon, Portugal, October 2004. University of Kent. (p. 210, 218)
- [204] Octavian Patrascoiu. YATL: Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Netherlands, January 2004. (p. 173)
- [205] Marian Petre and Thomas R. G. Green. Learning to read graphics: Some evidence that ‘seeing’ an information display is an acquired skill. *Journal of Visual Languages and Computing*, 4(1):55–70, 1993. (p. 57)
- [206] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998. relies on the concept of dusugaring, as a transformation from Haskell to a Core language. (p. 43)
- [207] Damien Pollet, Didier Vojtisek, and Jean-Marc Jézéquel. OCL as a core UML transformation language. WITUML 2002 Position paper, Malaga, Spain, June 2002. <http://ctp.di.fct.unl.pt/~ja/wituml02.htm>. (p. 24, 25)
- [208] Rajendra Raj and Yi-Min Wang, editors. *6th USENIX Conference on Object-Oriented Technologies and Systems*, San Antonio, Texas, USA, 2001. USENIX. (p. 212, 228)
- [209] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph rewriting – a constructive approach. *Electronic Notes in Theoretical Computer Science*, 2, 1995. (p. 182)
- [210] Edgars Rencis. Case study: UML to CSP transformation. In *AGTIVE 2007 Tool Contest Solutions* [213]. (p. 111)
- [211] A. Rensink and J. Warmer, editors. *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, Heidelberg, 2006. Springer Verlag. (p. 207, 213)

- [212] Arend Rensink. Time and space issues in the generation of graph transition systems. In *Proceedings of the International Workshop on Graph-Based Tools*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 127–139, Amsterdam, March 2005. Elsevier Science Publ. (p. 55)
- [213] Arend Rensink and Gabriele Tántzer. AGTIVE 2007 tool contest solutions. <http://www.se.eecs.uni-kassel.de/se/index.php?toolcontest>, 2007. (p. 209, 222)
- [214] Arend Rensink and Gabriele Tántzer. Graph transformation tools contest, at AGTIVE 2007. <http://www.coobra.cs.uni-kassel.de/index.php?toolcontest>, October 2007. (p. 42)
- [215] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In *ER'98: Proceedings of the 17th International Conference on Conceptual Modeling*, pages 449–464, London, UK, 1998. Springer-Verlag. (p. 16)
- [216] Robert Wagner. Consistency Management System for the Fujaba Tool Suite – MoTE/-MoRTen Plugins. <https://dsd-serv.uni-paderborn.de/projects/cms/>, January 2008. (p. 171, 191)
- [217] Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999. (p. 73, 74, 76)
- [218] David Roe, Krysia Broda, and Alessandra Russo. Mapping UML models incorporating OCL constraints into Object-Z. Technical Report 2003/9, Imperial College London, 2003. (p. 16)
- [219] Jose Raul Romero and Antonio Vallecillo. On the execution of odp computational specifications. *edocw*, 0:34, 2006. (p. 205)
- [220] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. (p. 136)
- [221] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. (p. 188, 210, 212, 224)
- [222] SAP. mySAP ERP. <http://www.sap.com/solutions/business-suite/erp/>, 2007. (p. 13)
- [223] Markus Scheidgen. CMOF-model semantics and language mapping for MOF 2.0 implementations. In *MBD-MOMPES'06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 84–93, Washington, DC, USA, 2006. IEEE Computer Society. (p. 206)
- [224] Hans Schippers. JMI Conforme Modeltransformator Generator. Master's thesis, University of Antwerp, Belgium, 2004. (p. 93, 101)

- [225] Hans Schippers and Pieter Van Gorp. Standardizing SDM for model transformations. In *Second Int. Fujaba Days (FD'04)*, Darmstadt, Germany, 2004. (p. 89)
- [226] Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):5–16, 2004. Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation. (p. 89)
- [227] A. Schürr, A. J. Winter, and A. Zündorf. *The PROGRES approach: language and environment*, pages 487–550. Volume II: Applications, Languages, and Tools of Ehrig et al. [70], 1999. discusses contracts like ensure $\Rightarrow$ DBC related, but procedural (graph operations in packages, no classes). (p. 182)
- [228] Andy Schürr. PROGRES: A visual language and environment for programming with graph rewrite systems. Technical Report AIB 94-11, RWTH Aachen, Fachgruppe Informatik, 1994. (p. 34)
- [229] Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG 1994*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1995. (p. 188, 199)
- [230] Andy Schürr. *Programmed Graph Replacement Systems*, pages 479–546. Volume I: Foundations of Rozenberg [221], 1997. (p. 182)
- [231] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Developing tools with the progres environment. In Nagl [180], pages 356–369. (p. 33)
- [232] Scott W. Ambler. A UML Profile for Data Modeling. <http://www.agiledata.org/essays/umlDataModelingProfile.html>, 2005. (p. 18)
- [233] E Seidewitz. What models mean. *IEEE Software*, 20, September 2003. (p. 13)
- [234] S. Sendall and W. Kozaczynski. Model Transformation - The Heart and Soul of Model-Driven Software Development. *IEEE Software, Special Issue on Model Driven Software Development*, pages 42–45, September 2003. (p. 41)
- [235] Keng Siau and Q. Cao. Unified modeling language (UML) – a complexity analysis. *Journal of Database Management*, 12, January 2001. (p. 16)
- [236] Anthony J. H. Simons and Ian Graham. 30 things that go wrong in object modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 221–242. Kluwer, Dordrecht, 1999. (p. 16)
- [237] Harry M. Sneed. Using XML to integrate existing software systems into the web. In *COMPSAC'02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 167–172, Washington, DC, USA, 2002. IEEE Computer Society. (p. 13)
- [238] Joel Spolsky. What is the work of dogs in this country? <http://www.joelonsoftware.com/articles/fog0000000012.html>, May 2001. (p. 156)

- [239] M. W. A. Steen, D. H. Akehurst, H. W. L. ter Doest, and M. M. Lankhorst. Supporting viewpoint-oriented enterprise architecture. In *Enterprise Distributed Object Computing Conference, (EDOC)*, pages 201–211, 2004. (p. 181)
- [240] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007*, volume 4735, pages 1–15. Springer LNCS, October 2007. (p. 38)
- [241] Perdita Stevens, Jon Whittle, and Grady Booch, editors. *UML'03: Proceedings of the 6th International Conference on The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, San Francisco, CA, USA, 2003. Springer-Verlag. (p. 220, 227)
- [242] Mirko Stölzel, Steffen Zschaler, and Leif Geiger. Integrating OCL and model transformations in Fujaba. *Electronic Communications of the European Association of Software Science and Technology (EASST)*, 5: OCL for (Meta-)Models in Multiple Application Domains(1863–2122), 2006. (p. 60, 101, 173)
- [243] Gerson Sunyé, François Pennaneac'h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML action semantics for executable modeling and beyond. In *CAiSE'01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, pages 433–447, London, UK, 2001. Springer-Verlag. (p. 24)
- [244] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In Martin Gogolla and Cris Kobryn, editors, *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes of Computer Science*, pages 134–148. IRISA, Springer Verlag, 2001. (p. 76)
- [245] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International Journal on STTT*, 6(3):203–218, August 2004. (p. 188)
- [246] Gabriele Tántzer and Arend Rensink. AGTIVE 2007 Tool Contest Overview. <http://www.informatik.uni-marburg.de/swt/active-contest/>, January 2008. (p. 108)
- [247] Gabriele Tántzer and Olga Runge. AGG – The Attributed Graph Grammar System: A Development Environment for Attributed Graph Transformation Systems. <http://tfs.cs.tu-berlin.de/agg/>, 2007. (p. 55, 182)
- [248] The Apache Software Foundation. Maven. <http://maven.apache.org/>, 2007. (p. 167)
- [249] Tom Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 1999. (p. 182)
- [250] Paolo Tonella and Alessandra Potrich. Reverse engineering of the interaction diagrams from c++ code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 159, Washington, DC, USA, 2003. IEEE Computer Society. (p. 43)

- [251] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2003. (p. 76)
- [252] Ragnhild Van Der Straeten, Tom Mens, and Viviane Jonckers. A formal approach to model refactoring and model refinement. *Software and Systems Modeling*, pages 139–162, June 2007. (p. 154)
- [253] Pieter Van Gorp. Generating JMI model transformation code from UML profile models for SDM. Demonstration at the SegraVis School on Foundations of Visual Modelling Techniques, May 2004. (p. 99)
- [254] Pieter Van Gorp. SDM transformatie voor method classes. In *FWO-MDE vergadering*. University of Antwerp, May 2005. <http://www.fots.ua.ac.be/~fots/research/fwo-mde-project/>. (p. 152)
- [255] Pieter Van Gorp. UML Profile for Data Modeling. <http://www.fots.ua.ac.be/~pvgorp/research/datamodelingprofile/>, 2005. (p. 18)
- [256] Pieter Van Gorp. Evaluation of the story driven modeling methodology: From towers to models. Technical report, University of Antwerp, Belgium, December 2007. (p. 33, 34, 36, 37)
- [257] Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Traceability and Fine-Grained Constraints in Interactive Inconsistency Management. In Tor Neple, Jon Oldevik, and Jan Aagedal, editors, *Second ECMDA Traceability Workshop*, July 2006. (p. 132, 173, 178, 180, 186, 193)
- [258] Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel. In *Proceedings of the 1st International Fujaba Days*, University of Kassel, Germany, Oct 2003. (p. 79, 81)
- [259] Pieter Van Gorp, Frank Altheide, and Dirk Janssens. Towards 2D Traceability in a Platform for Contract Aware Visual Transformations with Tolerated Inconsistencies. In Hung [118]. (p. 186, 196, 197, 200)
- [260] Pieter Van Gorp and Dirk Janssens. CAViT: a consistency maintenance framework based on visual model transformation and transformation contracts. In J. Cordy, R. Lämmel, and A. Winter, editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. (p. 139, 173)
- [261] Pieter Van Gorp, Dirk Janssens, and Tracy Gardner. Write once, deploy N: A performance oriented MDA case study. In David H. Akehurst, Marten van Sinderen, and Barrett R. Bryant, editors, *Enterprise Distributed Object Computing Conference, (EDOC)*, pages 123–134, Monterey, California, 20 September 2004. IEEE. (p. 173, 203)
- [262] Pieter Van Gorp and Tom Mens. A taxonomy of model transformation. <http://www.es.tu-darmstadt.de/english/events/tgg-workshop2006/presentations.html>. (p. 42)

- [263] Pieter Van Gorp and Olaf Muliawan. Standalone version of the MoTMoT UML2CSP solution. <http://www.fots.ua.ac.be/motmot/docs/uml2csp.php>, 2007. (p. 113)
- [264] Pieter Van Gorp, Olaf Muliawan, and Dirk Janssens. Hybrid transformation modeling: A case study based on developer interaction. In *1st International Triple Graph Grammar Workshop*, Bayreuth, Germany, September 2006. <http://www.es.tu-darmstadt.de/index2.php?page=2886>. (p. 185)
- [265] Pieter Van Gorp, Olaf Muliawan, Anne Keller, and Dirk Janssens. Executing a platform independent model of the UML-to-CSP transformation on a commercial platform. In Gabriele Tántzer and Arend Rensink, editors, *AGTIVE 2007 Tool Contest*, January 2008. (p. 107)
- [266] Pieter Van Gorp, Hans Schippers, Serge Demeyer, and Dirk Janssens. Students can get excited about formal methods: a model-driven course on petri-nets, metamodels and graph grammars. In Mirosław Staron, editor, *The 3rd Educators' Symposium of the 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Symposium Proceedings*, Göteborg, Sweden, 2007. IT University of Göteborg, Department of Applied Information Technology. (p. 42)
- [267] Pieter Van Gorp, Hans Schippers, and Dirk Janssens. Copying Subgraphs within Model Repositories. In Roberto Bruni and Dániel Varró, editors, *Fifth International Workshop on Graph Transformation and Visual Modeling Techniques*, Electronic Notes in Theoretical Computer Science, pages 127–139, Vienna, Austria, apr 2006. Elsevier. (p. 132, 135, 146, 152)
- [268] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Stevens et al. [241], pages 144–158. (p. 67, 68, 70, 71, 72, 75)
- [269] Filip van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques. In *International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) 2003*, 2003. (p. 71)
- [270] Bert Vanhooff, Stefan Van Baelen, Aram Hovsepyan, Wouter Joosen, and Yolande Berbers. Towards a transformation chain modeling language. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI)*, pages 39–48, 2006. (p. 44)
- [271] Daniel Varro, Mark Asztalos, Denes Bisztray, Artur Boronat, Rubino Geiss, Martin Gogolla, Pieter Van Gorp, Ole Kniemeyer, Anantha Narayanan, Edgars Rencis, and Erhard Weinell. Graph transformation tools contest on the transformation of UML models to CSP. In *3rd International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance (AGTiVE)*, Lecture Notes in Computer Science Proceedings (LNCS), Kassel, Germany, 2008. Springer. accepted for publication. (p. 107)
- [272] Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and

- Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 290–304. Springer, 2004. (p. 59)
- [273] Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57(2), 2001. (p. 42, 44)
- [274] Cory Wagner and Cory Janssen. Eat your own dog food. <http://www.investopedia.com/terms/e/eatyourowndogfood.asp>, December 2007. (p. 156)
- [275] Robert Wagner, Holger Giese, and Ulrich Nickel. A plug-in for flexible and incremental consistency management. In *Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development)*, San Francisco, USA, October 2003. (p. 93)
- [276] Don Wells and Laurie A. Williams, editors. *Proceedings of the Second XP Universe and First Agile Universe Conference (XP/Agile Universe 2002)*, volume 2418 of *Lecture Notes in Computer Science*. Springer, August 2002. (p. 68)
- [277] Kirsten N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109–142, 1997. (p. 57)
- [278] Jon Whittle. Workshop on consistency problems in UML-based software development – workshops and tutorials at the UML 2002 conference. In Jean-Marc Jézéquel, Heinrich Hussman, and Stephen Cook, editors, *UML'02: Proceedings of the 5th International Conference on The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 442–448, Dresden, Germany, 2002. Springer-Verlag. (p. 14)
- [279] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14:221–227, 1971. (p. 43)
- [280] David Wonnacott. Using accessory functions to generalize dynamic dispatch in single-dispatch object-oriented languages. In Raj and Wang [208], pages 93–102. (p. 59)
- [281] Albert Zündorf. From use cases to code – rigorous software development with UML. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 324–325, New York, NY, USA, 2001. ACM Press. (p. 33)
- [282] Albert Zündorf. Rigorous Object Oriented Software Development, 2002. Habilitation Thesis, University of Paderborn. Draft version 0.3. (p. 37, 89, 155, 199)
- [283] Albert Zündorf and Andy Schürr. Nondeterministic control structures for graph rewriting systems. In *WG'91: Proceedings of the 17th International Workshop*, pages 48–62, London, UK, 1992. Springer-Verlag. (p. 33)